Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware

Benjamin Camus, Thomas Paris, Julien Vaubourg, Yannick Presse, Christine Bourjot, Laurent Ciarletta, Vincent Chevrier

Co-simulation of Cyber-Physical Systems using a DEVS wrapping Strategy in the MECSYCO Middleware

Benjamin Camus*[,1], Thomas Paris[1], Julien Vaubourg[1], Yannick Presse[2], Christine Bourjot[1], Laurent Ciarletta[1], and Vincent Chevrier[1]

[1]Université de Lorraine, CNRS, Inria, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France.
[2]Inria, 54600 Villers-lès-Nancy, France

Abstract

Most modeling and simulation (M&S) questions about cyber-physical systems (CPS) require expert skills belonging to different scientific fields. The challenges are then to integrate each domain's tools (formalism and simulation software) within the rigorous framework of M&S process. To answer this issue, we give the specifications of the MECSYCO co-simulation middleware which enables to interconnect several pre-existing and heterogeneous M&S tools, so they can simulate a whole CPS together. The middleware performs the co-simulation in a parallel, decentralized and distributable fashion thanks to its modular multi-agent architecture. In order to rigorously integrate tools which use different formalisms, the co-simulation engine of MECSYCO is based on DEVS. The central idea of MECSYCO is to use a DEVS wrapping strategy to integrate each tool into the middleware. Thus, heterogeneous tools can be homogeneously co-simulated in the form of a DEVS system. By using DEVS, MECSYCO benefits from the numerous scientific works which have demonstrated the integrative power of this formalism and gives crucial guidelines to rigorously design wrappers. We demonstrate that our discrete framework can integrate a vast amount of continuous M&S tools by wrapping the FMI standard. To this end, we take advantage of DEVS efforts of the literature (namely, the DEV&DESS hybrid formalism and QSS solvers) to design DEVS wrappers for FMU components. As a side-effect, this wrapping is not restricted to MECSYCO but can be applied in any DEVS-based platform. We evaluate MECSYCO with the proof of concept of a smart-heating use-case, where we co-simulate non DEVS-centric M&S tools.

keywords: DEVS, co-simulation, FMI/FMU, QSS, DEV&DESS, hybrid modeling, parallel simulation, cyber-physical systems

*Corresponding author: benjamin.camus@inria.fr

1 Introduction

In this article, we are interesting in the modeling and simulation (M&S) of Cyber-Physical Systems (CPS). As defined by Rajkumar et al.[1], *"CPS are physical and engineered systems whose operations are monitored, coordinated, controlled and integrated by a computing and communication core."*. CPS can be for instance smart grids or autonomous cars.

By experimenting in a rigorous way on a simplification of a CPS (i.e. a model) instead of a real one, the M&S process avoids cost, time and ethic constraints, and thus positions itself as a choice tool for the CPS science. However, when applied in this context, the M&S process faces many specific challenges. Indeed, the expert skills required for describing a CPS may come from different domains (e.g. for a smart grid: telecommunications, information systems, electrical grids), each of them having their own well-tried and well-tested models and M&S tools (i.e. formalisms and simulation software). **The challenges are then to reconcile these heterogeneous points of view, and to integrate the models and tools of each domain within the rigorous framework of the M&S process.**

A very promising strategy to tackle these challenges lies in co-simulation. Co-simulation consists in performing a simulation by reusing models implemented in different simulation software, managing exchanges of data between these software, and synchronizing their execution in order to make their models interact. It allows every specialists involved in the M&S process of a CPS to keep using the tools which are popular in his/her community while providing to each of them a multidisciplinary context. In addition, every simulator can (in some cases) be executed on an individual machine, enabling to co-simulate very large systems. However, co-simulations face many issues directly related to the heterogeneity of models and tools that need to interact together.

Our contribution to tackle these issues is twofold in this paper.

We give the whole operational specification of MECSYCO (Multi-agent Environment for Complex-SYstem CO-simulation). MECSYCO is a middleware dedicated to the co-simulation of CPS i.e. it enables to interconnect several pre-existing and heterogeneous (both at the software and formal levels) M&S tools belonging to different scientific fields, so they can simulate a whole CPS together. The co-simulation engine of MECSYCO is based on the DEVS formalism in order to integrate tools which use different modeling formalisms. Each tool is integrated into the middleware thanks to a DEVS wrapping strategy: a wrapper must be designed in order to control the tool like a DEVS simulator. Thus, heterogeneous tools can be homogeneously co-simulated in the form of a DEVS system. The choice of using DEVS as a pivotal formalism is motivated by numerous scientific works which have demonstrated over the years that DEVS can rigorously integrate many other M&S formalisms. A very important practical advantage is that these works also describe *how* each formalism can be integrated into DEVS, thus giving crucial guidelines and tools for rigorously designing the MECSYCO wrappers.

We demonstrate that this approach can rigorously integrate equation-based continuous tools and make them interact with discrete-event models. We take advantage of DEVS efforts of the literature (namely, the DEV&DESS hybrid formalism and the QSS solver strategy) to design DEVS wrappers for continuous tools. We exploit the federating FMI (Functional Mockup Interface) standard in order to make our wrappers compliant with a vast amount of tools. Thus, we propose DEVS wrappers for FMU components.

We illustrate these two contributions with a proof of concept of a smart-heating use-case, where we integrate and co-simulate non DEVS-centric M&S tools, namely OpenModelica and NS-3.

The paper is organized as follows. Section 2 presents the different challenges related to the co-simulation of CPS and existing solutions in the literature. In order to make our proposition fully understandable for non-specialist readers, we introduce in Section 3 the background and concepts (i.e. DEVS and FMI) we used. Section 4 details our global proposal and discusses our positioning with regards to the literature. The Section 5 presents our first contribution: the MECSYCO platform which enables the parallel co-simulation of CPS in a rigorous and decentralized way. Sections 6, 7 and 8 explain our second contribution: the DEVS wrapping of the FMI standard. In Section 9, we discuss the strengths and drawbacks of our solution. Finally, in Section 10 we illustrate our proposition with a smart heating use case.

2 Co-simulation Challenges and Related Works

When co-simulating a CPS, the system is represented as a set of interacting subsystems. Each of them is modeled separately, possibly with different tools (software and formalisms). Co-simulating consists in managing the synchronization of these heterogeneous simulators as well as the exchange of data between them. This raises two major challenges presented below.

2.1 Simulation Software Interoperability

From a software perspective, co-simulation implies dealing with a heterogeneous set of simulation software. Indeed, as shown in Table 1, different domains of expertise may have different simulation software, potentially implemented in different programming languages and compliant with different operating systems (OS). Moreover some of these simulation software must be available only on some specific hardware (e.g. when a private license is required). Interoperability processes are then required[2] to synchronize these heterogeneous software executions and manage exchanges of usable data between them [3].

This interoperability can be achieved in an ad-hoc way by directly modifying simulation software to make them compliant with each other. A more generic solution consist in using a simulation middleware dedicated to the management of the interoperability within the co-simulation. The advantage of this solution is that it is flexible: we can easily adding, removing and changing some simulation software without impacting the rest of co-simulation implementation. This is feasible because in this case, simulation software do not have to be directly interoperable with each other, but have to be interoperable with the middleware instead. The co-simulation middleware can also serve as a communication middleware, enabling the distribution of the co-simulation and the compliance with the required hardware and OS diversity.

The High Level Architecture (HLA) standard [8] gives generic guidelines and rules for building an event-based co-simulation middleware. However, HLA does not give the whole specification of the co-simulation middleware. Hence, HLA does not detail the (parallel or sequential) synchronization algorithm, the distributed architecture and its implementation and let them be tool-specific. As a drawback, simulation from one platform to another may be not reproducible, and different implementations of HLA may be not interoperable and therefore can not be simultaneously used in a co-simulation. Other co-simulation middleware such as Mosaik [9] are based on a discrete time-step framework which does not enable the rigorous integration of

Table 1: Example of M&S application domains and their simulation software.

Domain	Simulation software	Languages	Operating system
Collective motion	NetLogo [4]	Java API Java & Scala	GNU/Linux, Windows, Mac OS
	GAMA [5]	Java	GNU/Linux, Windows, Mac OS
Telecom networks	NS-3 [6]	C++, Python API	GNU/Linux
	OMNeT++ [7]	C++	GNU/Linux, Windows, Mac OS
Robotic	VREP	C/C++, Lua, Python, Java	GNU/Linux, Windows, Mac OS
Physical system	Dymola	Proprietary code	Windows
	Matlab/Simulink	Proprietary code, C/C++ API, Fortran	GNU/Linux, Windows, Mac OS

models written in heterogeneous formalisms.

2.2 Multi-Formalism Integration

Because of its own heterogeneity, a CPS may exhibit both discrete and continuous dynamics, and several formalisms may be required to describe the whole system [10]. For example, the cyber part is traditionally discrete whereas the physical one is rather continuous. Formalisms can be for instance differential or algebraic equations for the continuous parts, but event-based, finite-state automata or time-stepped models for the discrete parts.

As a consequence, discrete and continuous models may interact and co-evolve inside a same co-simulation. At the execution level, this formalism heterogeneity implies dealing with different scheduling policies: cyclic or variable time-steps, event-based, etc. A rigorous framework is then needed to integrate these different models in order to have an univocal behavior of the co-simulation [11].

Two solutions exist to integrate these different formalisms[10]:

Translate the models in a same formalism and perform the simulation using the abstract simulator of this formalism. This is the solution chosen by $AToM^3$ [12], which enables to automatically translate two models, using a sequence of transformations, to their closest common formalism. To do so, $AToM^3$ relies on a Formalism Transformation Graph where every node corresponds to a formalism and each arc represents an existing automatic translation. The shortcoming of this solution is that it forces to rewrite and re-implement the existing models. Thus, it does not have the advantages of co-simulation –i.e. it requires translation and implementation efforts (when not automatic) which may introduce errors.

Use a hybrid M&S formalism which explicitly describes how continuous and discrete systems interact and co-evolve. This super-formalism can be DEV&DESS [13] or HFSS[14] (Heterogeneous Flow System Specification). Both of them merge a whole set of traditional techniques used in the field of hybrid modeling. Such techniques notably include (1) the integration of discrete input events during the evolution of the continuous system, and (2) the generation of two kinds of discrete-events during the simulation: time-events and state-events[11] generated from the continuous system state. While the former consist in events scheduled at predefined simulation times, the latter corresponds to events whose occurrences are related to some specific conditions on the continuous state (usually when a continuous variable crosses a given threshold). From a simulation perspective, the challenge is to integrate this discrete-event logic, in a generic way, during the numerical resolution of the continuous system (which is concerned with finding the best trade-off between the accuracy of the solution and the simulation efficiency[15]). Most notably, the detection and the accurate localization in time of state-events during the simulation is a well-known issue in hybrid simulation [16].

2.3 Synthesis

To sum up, setting up a co-simulation requires to solve a set of specific issues at the formalism and the software levels. The solutions to provide are directly related to the heterogeneity found at each of these levels.

Additionally in a M&S process, modularity is often required –i.e. to be able to add/remove/change models or simulation software and their connections without redefining all the co-simulation from scratch [17].

In order to fulfill these requirements, ad-hoc solutions should be avoided and a more generic and rigorous framework is needed. In the following, we detail the background and concepts used to meet these requirements.

3 Background and Concepts

Our proposal relies both on the DEVS formalism and the FMI standard. In this section, we describe them in order to make our proposition fully understandable for non-specialist reader.

3.1 DEVS Formalism

DEVS [18] is an event-based formalism for the M&S of system of systems. One important feature of DEVS is its integrative power for multi-paradigm M&S [19]. Indeed, not only DEVS appears to be universal for describing discrete-event systems [18], but it can also integrate continuous systems [20] expressed for instance with differential equations [21]. Of particular interest in the scope of this article is the fact that, as shown by Zeigler[22], DEVS can also embed the DEV&DESS formalism [13]. This formalism offers a sound framework for representing hybrid systems as it describes how continuous systems interact and co-evolve with the discrete world.

Besides, DEVS can encapsulate differential and algebraic equation solvers by relying on a quantized integrator approach like the Quantized State Systems (QSS) method [23]. This approach is based on state quantization instead of the time discretization used by traditional integration methods. This strategy shows in some cases [24] better performances than traditional methods [25]. QSS is well-suited for hybrid modeling as it makes the continuous component equivalent to a DEVS model, which naturally integrates input events, and makes state-events detection trivial and costless [26].

As summarized by Quesnel[21], the integration of a formalism in DEVS can be performed either by a mapping or a wrapping. While the former consists in establishing the equivalence between the formalisms, the latter implies bridging the gap between the two abstract simulators [27]. The advantage of the wrapping strategy is to enable reusing pre-existing models already implemented in some simulation software [28].

The following part is a formal description of DEVS in order to fully understand our proposal, especially concerning the wrapping of continuous models.

DEVS distinguishes atomic from coupled models. A DEVS atomic model i describes the behavior of the system and corresponds to this structure:

$$M_i = (X_i, Y_i, S, \delta_{ext}, \delta_{int}, \lambda, ta) \quad (1)$$

where:

$X_i = \{(p, v) | p \in InPorts_i, v \in V_{X_i}\}$ is the set of input ports and values. These ports can receive external input events,

$Y_i = \{(p, v) | p \in OutPorts_i, v \in V_{Y_i}\}$ is the set of output ports and values. These ports can send external output events,

S is the set of the model states,

$\delta_{ext} : Q \times X_i \to S$ is the external transition function (describing how the model reacts to input events) where

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state of the model,

e is the elapsed time since the last transition,

$\delta_{int} : S \to S$ is the internal transition function describing the internal dynamic of the model –i.e. the function processes an internal event which changes the model state,

$\lambda : S \to Y_i$ is the output function describing the output events of the model according to its current state,

$ta : S \to \mathbb{R}^+_{0,\infty}$ is the time advance function describing how long the model will stay in the same current state (in the absence of input event). The function is used to get the date of the next internal event.

A coupled model describes the structure of the system. It corresponds to the following structure, describing a set of interconnected atomic models:

$$N = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC) \quad (2)$$

where:

$X = \{(p, v) | p \in InPorts, v \in V_{X_i}\}$ is the set of input ports and values

$Y = \{(p, v) | p \in OutPorts, v \in V_{Y_i}\}$ is the set of output ports and values,

D is the set of models id,

$EIC = \{((N, ip_N), (d, ip_d)) | ip_N \in InPorts, d \in D, ip_d \in InPorts_d\}$ is the set of external input couplings,

$EOC = \{((d, op_d), (N, op_N)) | op_N \in OutPorts, d \in D, op_d \in OutPorts_d\}$ is the set of external output couplings,

$IC = \{((a, op_a), (b, ip_b)) | a, b \in D, op_a \in OutPorts_a, ip_b \in InPorts_b\}$ is the set of internal couplings,

The closure under the coupling of DEVS is an important property which enables hierarchical modeling by proving that a coupled model is equivalent to an atomic one. Therefore, a DEVS coupled model can be composed of a set of interconnected atomic and coupled models (these latter may be at their turn composed of coupled models, etc.). DEVS proposes sequential and parallel abstract simulators and coordinators for respectively simulating atomic and coupled models. Thanks to the closure under the coupling of DEVS, these abstract simulators and coordinators can be controlled in a unified way using the DEVS simulation protocol.

3.2 The FMI Standard

FMI [29] is a standard of the MODELISAR Consortium and the Modelica Association which proposes a generic software interface for manipulating equation-based models and their solvers. These models may be composed of a mixture of differential, algebraic and discrete-time equations. FMI aims at (1) defining a generic way of exchanging and using models designed with different equation-based simulation tools, and (2) protecting the intellectual property of these models by ensuring that they are seen as black-boxes.

A model implementing the FMI standard is called a Functional Mock-up Unit (FMU). The FMU interface differentiates the output variables whose values are accessible from the outside (i.e. equivalent to output ports of the model), from the input variables whose values can be set from the outside (i.e. equivalent to input ports of the model). From a software perspective, this interface is composed of a set of C functions, and an XML file. C functions enable controlling the FMU, whereas the XML file describes the FMU and its interface. More precisely, the XML file describes names, types (i.e. Real/Integer/Boolean/String), variability (constant/discrete/-continuous) and causality (input/output/parameter) of the variables, as well as the continuous states vector.

So far around 52 simulation tools (e.g. Dymola, MATLAB/Simulink, OpenModelica) claim to be compliant with FMI v2.0 (80 with FMI v1.0), including 23 tools officially certified (29 with FMI v1.0)[1]. Several of these tools are based on Modelica [30] which is an object-oriented language adapted to the modeling and simulation of hybrid systems. In order to support the standard, a tool need either (1) to be able to export its own models as FMUs, or (2) to be able to import existing FMUs and use them as components in models. FMI allows two ways of exporting and importing a model: FMI for co-simulation (FMI-CS) and FMI for model-exchange (FMI-ME).

With **FMI-ME**, the model is exported without its solver. The FMU must be then associated with an external solver in order to be simulated. For that purpose, the solver can especially use the following C functions of the FMU API:

- `fmi2GetReal/Integer/Boolean/String` returns the current value of a given output variable.
- `fmi2SetReal/Integer/Boolean/String` sets a specific input variable to a given value.
- `fmi2SetTime` sets the clock of the model to a given simulated time.
- `fmi2GetContinuousStates` returns the continuous state vector.
- `fmi2SetContinuousStates` sets a continuous state vector.
- `fmi2GetDerivatives` returns the derivative vector of the continuous state.
- `fmi2CompletedIntegratorStep` indicates that the integration step is finished, and evaluates if internal events have to be processed.
- `fmi2GetEventIndicators` returns indicators of state-events occurrences.
- `fmi2EnterEventMode` enters into the discrete event mode, i.e. makes the discrete-time equations active. While the FMU is in this mode, the integration of the continuous state is stopped but discrete-events (time, state or external) can be processed.
- `fmi2EnterContinuousTimeMode` enters into the continuous-time mode, i.e. disable the discrete-time equations. In this mode, the continuous state of the FMU can be solved, but the discrete state has to remain constant (i.e. events can not be processed).
- `fmi2NewDiscreteStates` evaluates the discrete-time equations (should therefore only be called in event mode) –i.e. processes the potential time and state events. Information returned by this function includes (1) the date of the next time-event (when scheduled), (2) indication if the processed event(s) has changed the continuous state (creating a discontinuity in the state trajectory), and (3) indication if the discrete state has to be immediately re-evaluated (e.g. to solve an internal algebraic loop).

With **FMI-CS**, a model is exported with its solver. As this solver has a passive behavior, an FMU for co-simulation (FMU-CS) is considered as a slave, and proposes in particular the following C functions in order to be controlled by a master algorithm[31]:

- `fmi2DoStep` performs an integration for a given duration.
- `fmi2SetReal/Integer/Boolean/String` sets a specific input variable to a given value.
- `fmi2GetReal/Integer/Boolean/String` gets the current value of a given output variable.
- `fmi2GetFMUState` and `fmi2SetFMUState` are optional (but essential[32]) functions used to export/import the model state. They enable to perform a rollback during the simulation of the model.

[1] According to https://www.fmi-standard.org/tools on 2/16/2017.

FMI gives generic guidelines on how a master should manage a set of interconnected FMUs in order to jointly solve their equations: FMU executions are synchronized thanks to communication points. These communication points, shared by every involved FMU, correspond to points in the simulated time where (1) the FMU simulation must be stopped, and (2) exchanges of data have to be performed between FMUs according to their output-to-input links.

Aside from these guidelines, FMI does not give specifications for a master algorithm. Consequently, different master algorithms are currently developed, like FIDE[33] (FMI Integrated Design Environment) and DACCOSIM[34] (Distributed Architecture for Controlled CO-SIMulation). Numerous issues related to the distributed numerical resolution of the system[32] are still under investigation by the community (e.g. How to determine the best communication point interval during the simulation? How to manage algebraic loop between FMUs?).

4 Proposal and Positioning

On the one hand, a co-simulation middleware is required to manage the interoperability of different M&S tools. On the other hand, we need a formal solution to rigorously integrate heterogeneous formalisms.

We propose to tackle these two requirements by defining a modular co-simulation middleware called MECSYCO. We integrate tools which are formally heterogeneous by using DEVS as a pivotal formalism in the following way:

1. Integrate the different tools in DEVS by using a wrapping strategy –i.e. instead of directly writing or transforming the models in DEVS, provide additional mechanisms in order to bridge the gap between the tools and the DEVS abstract simulator. Hence, each tool can be controlled like a DEVS simulator and we do not have to implement the model again.

2. Connect these wrapped tools within a DEVS coupled model.

3. Simulate the DEVS coupled model using the DEVS simulation protocol, in order to perform a co-simulation in an unified way.

We choose DEVS because of its striking capacity to integrate the formal heterogeneity of a co-simulation. Other M&S formalisms which enables the integration of continuous and discrete dynamics could have been used. For instance the HFSS formalism provide several interesting properties for hybrid system modeling (such as the dynamic structure[14]), and has some advantages over DEVS (e.g. the possibility to represent geometrical solvers [35]). However, DEVS benefits from a greater amount of scientific works which have demonstrated over the years its integrative power. These works are essential in the context of co-simulation because they also describe *how* each specific M&S formalism should be integrated in a rigorous way, thus giving crucial guidelines and tools for rigorously designing our wrappers.

So far, we successfully defined DEVS wrappers for discrete modeling tools like the MAS simulator NetLogo [36], and the telecommunication network simulators NS-3 and OMNeT++/INET [37]. Aside from several difficulties met when wrapping NS-3 and OMNeT++/INET (mainly due to the high level of modeling details offered by these platforms, as well as to the complexity of the opening and the distribution of their telecommunication models), making these discrete modeling tools compliant with the DEVS simulation protocol was a straightforward process. The reason is that these platforms have a discrete modeling paradigm very close to DEVS.

However, according to our experience with MECSYCO, several difficulties may arise when wrapping a simulation tool in DEVS. These problems depend mainly on two criteria:

- **The M&S formalism used by the tools** may not be explicitly defined by the software specifications, and/or may be very different from DEVS. Accordingly, the challenge is to answer the questions: what is the formalism used by the tools? How to bridge the formal gap between this formalism and DEVS?

- **The software interface** with the middleware may be difficult to produce as the tools API and the software architecture are not always documented and fully compliant with the DEVS simulation protocol. Moreover, the software may not be conceived to be externally manipulated.

Things getting especially complex with equation-based tools as their continuous modeling paradigm is very different from the discrete DEVS one. Thus, we need to bridge the gap between the discrete and the continuous paradigms, and a more complex wrapping strategy based on the hybrid capacity of DEVS is required. Regarding this issue, wrapping each of these equation-based tools (e.g. OpenModelica, Dymola, Matlab/Simulink) separately would be very inefficient.

However, most of these tools are compatible with the FMI standard which brings a generic API to manipulate equation-based models and their solvers. **Thus, in order to integrate continuous tools into MECSYCO in the more generically possible way at the software level, we propose to define DEVS wrappers for the FMI standard.** We base this wrapping on the DEV&DESS formalism to handle the interactions between the continuous

equation-based model and the discrete event paradigm of DEVS. Since FMUs for co-simulation and FMUs for model exchange do not have the same API and do not convey the same constraints, we specify a different wrapper for each of them in order to be fully compliant with the standard. Thanks to these wrappers, continuous equation-based models are integrated in MECSYCO in the following way:

1. The model is specified in a dedicated equation-based tool (such as OpenModelica).
2. The model is exported as an FMU using the built-in export features of the tool[2].
3. This FMU is linked to the wrapper and integrated in the co-simulation.

Please note that rather than focusing on the distributed numerical resolution aspects which arise when several FMUs are directly interconnected, we focus in this paper on the hybrid simulation issues which arise when an FMU interacts with a discrete-event component (e.g. a NS-3 model). Indeed, in a hybrid context, the communication points simulation strategy of FMI faces the following issues:

- State events occurring between two points of communication are localized at the upper communication point, pending improvements of the hybrid co-simulation in the FMI standard.
- New inputs are only taken into account at the next communication point, no matter when they are received.

As a result, an effort is required to integrate the operational software in such a way as to respond to events.

To summarize, our proposition is twofold and can be seen at two levels of detail. On the most generic level, MECSYCO is a co-simulation middleware which focuses on the formal integration of pre-implemented models by using a DEVS based wrapping strategy. This strategy is supported by all the integrative work around DEVS [10]. In this way, our proposition responds to both the formal integration and the software interoperability requirements of CPS co-simulations (detailed in Section 2). On a more specific level, we propose a way to integrate equation-based continuous tools into MECSYCO by defining DEVS wrappers based on the hybrid formalism DEV&DESS and the QSS solver strategy. We use the emerging standard FMI as a generic way to integrate continuous models at the software level. It is important to note that our purpose here is more focused on the rigorous integration of heterogeneity in co-simulations rather than the co-simulations efficiency.

[2]Consequently, this feature is mandatory. When not available, only ad-hoc wrappings are possible.

Comparing to the related works of the literature, the focus on reuse of pre-existing models distinguishes our proposal from multi-paradigm approach like $AToM^3$ (see Section 2.2). Unlike the other DEVS based tools of the literature (like VLE[38]) whose primary purpose is to design and simulate models in DEVS, our platform is dedicated to the DEVS wrapping and the co-simulation of already existing models and simulators. In contrast to HLA (see Section 2.1), the formal integration of MECSYCO is driven by DEVS wrapping. We also specify the whole software architecture and synchronization algorithm (Section 5.3) making two implementations of MECSYCO interoperable. In contrast to the Mosaik co-simulation middleware [9], we can integrate rigorously M&S tools which are formally heterogeneous thanks to our DEVS framework. Furthermore, in contrast with other master algorithms which are dedicated to the co-simulation of FMU components (e.g DACCOSIM [34]), MECSYCO is not limited to a specific simulation software or norm.

5 The MECSYCO Platform

5.1 Generalities

MECSYCO [36] is a middleware dedicated to the co-simulation of Cyber-Physical Systems (CPS) that enables to interconnect several pre-existing and heterogeneous (both at the software and formal levels) M&S tools. For this purpose, MECSYCO manages the data exchanges between these tools, and synchronizes their executions in a parallel and fully decentralized way.

The co-simulation engine of MECSYCO is based on the DEVS formalism in order to integrate tools which use different modeling formalisms (e.g. discrete-event, ODE). Each tool is integrated into the middleware thanks to a DEVS wrapping strategy: a wrapper must be designed so the tool can be controlled like a DEVS simulator. Thus, heterogeneous tools can be homogeneously co-simulated in the form of a DEVS system.

MECSYCO is based on the AA4MM (Agents & Artifacts for Multi-Modeling) paradigm [39] (from an original idea of Bonneaud [40]), proposing to see an heterogeneous co-simulation as a multi-agent system. Within this scope, each couple model/simulator corresponds to an agent, and the data exchanges between the simulators correspond to the interactions between the agents. Thus, the co-simulation of the system corresponds to the dynamics of interaction between agents. Agent autonomy enables encapsulating legacy software by the use of wrappers[41]. Originality with regard to other multi-agent multi-model approaches is to consider the interactions in an indirect way thanks to the concept of passive computational entities called artifacts [42].

MECSYCO implements the AA4MM concepts according to the DEVS simulation protocol for coordi-

nating the executions of the simulators and managing interactions between models. By following the multi-agent paradigm from the concepts to their implementation, MECSYCO ensures a modular, extensible (i.e. features can be easily added such as an observation system) decentralized and distributable parallel co-simulation. The MECSYCO middleware is completely modular and can be distributed on several machines which may run on different OS (e.g. GNU/Linux, Mac OS, Microsoft Windows). It is currently used to study green cloud computing [43] and for the M&S of smart electrical grids in the context of a partnership between LORIA/Inria[3] and EDF R&D (main French electric utility company) [44].

In the following, we describe these concepts and their specifications.

5.2 MECSYCO Concepts

MECSYCO relies on four concepts to describe a co-simulation.

A **model** m_i is a partial representation of the target system implemented in a simulation software s_i (cf. corresponding symbol in Figure 1a). A model has a set of input ports $x_i^{1..n}$ and output ports $y_i^{1..m}$.

An **m-agent** $\mathcal{A}_i$ (cf. corresponding symbol in Figure 1b) manages a model m_i and is in charge of the interactions of this model with the other ones. Therefore, the m-agent is equivalent to a parallel abstract simulator for the models.

Each m-agent $\mathcal{A}_i$ sees its model m_i as a DEVS atomic model thanks to its **model artifact** $\mathcal{I}_i$ (cf. corresponding symbol in Figure 1d). Therefore, $\mathcal{I}_i$ acts as a DEVS wrapper for m_i - i.e. it implements the DEVS simulation protocol functions for controlling m_i evolution through s_i.

Each interaction from an m-agent $\mathcal{A}_i$ to an m-agent $\mathcal{A}_j$ is reified by a **coupling artifact** $\mathcal{C}_j^i$ (cf. corresponding symbol in Figure 1c). A coupling artifact $\mathcal{C}_j^i$ works like a mailbox: the artifact has a buffer of events where the m-agents can post their external output events and get their external input events. Accordingly, a coupling artifact plays two roles: for $\mathcal{A}_i$, it is an **output coupling artifact**, whereas for $\mathcal{A}_j$ it is an **input coupling artifact**. Coupling artifacts can transform data exchanged between the models using operations that can be for instance, spatial and time scaling operations (e.g. converting kilometers to meters or hours to minutes).

According to the multi-agent paradigm, m-agents only have a local knowledge of the coupled model interconnections. The set of internal couplings between coupled model IC is split such as an m-agent $\mathcal{A}_i$ only knows which input coupling artifacts correspond to its model input ports, and which output coupling artifacts correspond to its model output ports. We define the set of input links IN_i of $\mathcal{A}_i$ as being composed of the couples (j,k) mapping the input coupling artifact $\mathcal{C}_i^j$ with the input port x_i^k. We define the set of output links OUT_i of $\mathcal{A}_i$ as being composed of the couples (n,j) mapping the output port y_i^n with the output coupling artifact $\mathcal{C}_j^i$.

The connection of the output ports of a model m_i with the input ports of a model m_j is done by the coupling artifact $\mathcal{C}_i^j$. The link from a model m_i to a model m_j (noted as L_j^i) corresponds to the tuple $(n,k,o_{j,k}^{i,n})$. It maps the output port y_i^n with the input port x_j^k and applies the o_k^n operation to transform the event between these two models representation. By default, an operation corresponds to the identity operation id. The Table 2 and the Figure 3 illustrate how a DEVS coupled model (showed in Figure 2) is described in a decentralized and distributable way thanks to MECSYCO.

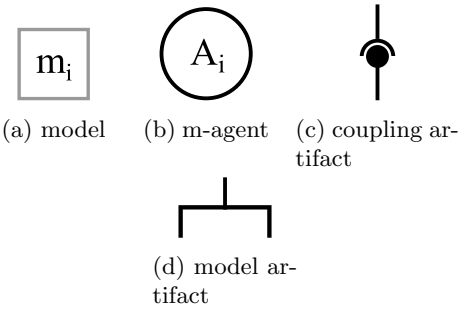


Figure 1: Symbols of the MECSYCO components.

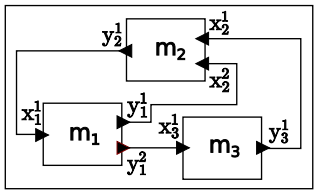


Figure 2: Bloc diagram view of a DEVS coupled model.

5.3 Operational Specifications

The behavior of each m-agent corresponds to the DEVS conservative parallel abstract simulator, based

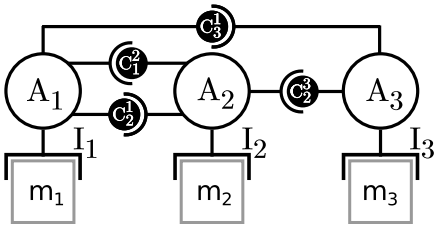


Figure 3: Graphical representation of the MECSYCO co-simulation of Table 2.

[3]French computer science research institutes.

Table 2: Decentralized MECSYCO co-simulation of the DEVS coupled model of Figure 2.

Descriptions	**Notations**
Output links of m_1	$\mathsf{OUT}_1 = \{(1,2),(2,3)\}$
Input links of m_1	$\mathsf{IN}_1 = \{(2,1)\}$
Output links of m_2	$\mathsf{OUT}_2 = \{(1,1)\}$
Input links of m_2	$\mathsf{IN}_2 = \{(1,2),(3,1)\}$
Output links of m_3	$\mathsf{OUT}_3 = \{(1,2)\}$
Input links of m_3	$\mathsf{IN}_3 = \{(1,1)\}$
Links from m_1 to m_2	$\mathsf{L}_2^1 = \{(1,2,o_{2,2}^{1,1})\}$
Links from m_1 to m_3	$\mathsf{L}_3^1 = \{(2,1,o_{3,1}^{1,2})\}$
Links from m_2 to m_1	$\mathsf{L}_1^2 = \{(1,1,o_{1,1}^{2,1})\}$
Links from m_3 to m_2	$\mathsf{L}_2^3 = \{(1,1,o_{2,1}^{3,1})\}$

on the Chandy-Misra-Bryant (CMB) algorithm [45, 46]. This algorithm is proven to be deadlock free and to respect the causality constraint [18] –i.e. to ensure that the *"execution of the simulation program on a parallel computer will produce exactly the same results as an execution on a sequential computer"* [47].

As we focus on the rigorous formal integration instead of performances, a conservative algorithm is chosen because it does not impose specific ability like rollback to the model we want to integrate.

Within this behavior, each m-agent $\mathcal{A}_i$ shares its Earliest Output Time estimate noted EOT_i in its environment. EOT_i corresponds to the date (in simulation time), below which $\mathcal{A}_i$ guarantees it will not send new external output events. $\mathcal{A}_i$ shares EOT_i as the **link time** of each of its output coupling artifact. The link time of a coupling artifact $\mathcal{C}_j^i$ is noted LT_j^i and corresponds to the simulated time (initially equals to 0) up to which $\mathcal{A}_i$ has simulated the links from m_i to m_j [45].

Every m-agent $\mathcal{A}_i$ uses the link times of all of its input coupling artifacts to compute its Earliest Input Time estimate noted EIT_i. This EIT_i corresponds to the date (in simulated time) below which $\mathcal{A}_i$ will not receive any new external input event. EIT_i corresponds to the minimum link time of all of $\mathcal{A}_i$'s input coupling artifacts.

For each m-agent $\mathcal{A}_i$, all events (internal or external) with a timestamp inferior or equals to EIT_i are said to be safe to process. In order to fulfill the causality constraint, each m-agent must process only safe events and in an increasing timestamped order.

Each EOT_i is given by the $\mathtt{Lookahead}_i$ function:

$$Lookahead_i() = min\{nt_i, \mathsf{EIT}_i + D_i, t_{in_i+D_i}\} \quad (3)$$

with nt_i the next internal event time of m_i, t_{in_i} the time of the earliest event waiting to be processed in $\mathcal{A}_i$'s input coupling artifact, and $D_i (D_i > 0)$ the minimum propagation delay of m_i. This minimum propagation delay corresponds to the minimum delay (in simulated time) below which the processing of an external event can not schedule a new internal event in a model m_i. D_i has to be determined for each model m_i in the co-simulation.

This behavior, enabling to simulate a model until a time Z, is formalized within the MECSYCO paradigm by the Algorithm 1. This algorithm is based on the artifact specifications detailed below.

A coupling artifact $\mathcal{C}_j^i$ proposes six functions to $\mathcal{A}_i$ and $\mathcal{A}_j$:

- `post(`e_{out}^n`,` t_i`)` stores in the artifact buffer and transforms (according to the $\mathcal{C}_j^i$ operation) the external output event e_{out}^n generated at the (simulated) time t_i trough the output port y_i^n.
- `getEarliestEvent(`k`)` returns the earliest external input event for the k^{th} input port of m_j, x_j^k.
- `getEarliestEventTime(`k`)` returns the time of the earliest external event for x_j^k.
- `removeEarliestEvent(`k`)` removes the earliest external event for x_j^k, from the artifact buffer.
- `setLinkTime(`t_i`)` sets LT_j^i to t_i.
- `getLinkTime()` returns LT_j^i.

In order to manipulate m_i, each model artifact $\mathcal{I}_i$ proposes the following DEVS simulation protocol functions to $\mathcal{A}_i$ (they have to be defined for each simulation software):

- `init()` initializes the model m_i. It sets the parameters and the initial state of the model.
- `processExternalEvent(`e_{in_i}`,`t_i`,`x_i^k`)` processes the external input event e_{in_i} at simulation time t_i in the k^{th} input port of m_i, x_i^k.
- `processInternalEvent(`t_i`)` processes the internal event of the model m_i scheduled at time t_i.
- `getOutputEvent(`y_i^n`)` returns $e_{out_i}^n$, the external output event at the n^{th} output port of m_i, y_i^n.
- `getNextInternalEventTime()` returns the time of the earliest scheduled internal event of the model m_i.

5.4 Implementation

MECSYCO is currently implemented in Java (available at `http://mecsyco.com` under AGPL) and C++. In order to make these two versions interoperable together and to perform distributed co-simulations, MECSYCO relies on the JSON format and the OpenSplice implementation of the OMG standard DDS (Data Distribution Service). Using Opensplice, coupling artifacts are divided into two parts – reader and writer – in order

Algorithm 1 $\mathcal{A}_i$ m-agent's behavior.

INPUT: IN_i, OUT_i, Dt_i
OUTPUT:

$nt_i \leftarrow \mathcal{I}_i$.getNextEventTime()
$t_{in_i} \leftarrow +\infty$
$\mathsf{EOT}_i \leftarrow 0$
$\mathsf{EIT}_i \leftarrow 0$
▷ While the end of simulation.
while $(\neg endOfSimulation)$ **do**
  $\mathsf{EIT}_i \leftarrow +\infty$
  $t_{in_i} \leftarrow +\infty$
  for all $(j,k) \in \mathsf{IN}_i$ **do**
    if $\mathcal{C}_i^j$.getLinkTime() $< \mathsf{EIT}_i$ **then** ▷ Compute EIT_i.
      $\mathsf{EIT}_i \leftarrow \mathcal{C}_i^j$.getLinkTime()
    end if
    if $\mathcal{C}_i^j$.getEarliestEventTime(k) $< t_{in_i}$ **then** ▷ Take the next external event.
      $t_{in_i} \leftarrow \mathcal{C}_i^j$.getEarliestEventTime($k$)
      $ein_i \leftarrow \mathcal{C}_i^j$.getEarliestEvent($k$)
      $p \leftarrow k$ ▷ Save the corresponding input port.
      $c \leftarrow j$ ▷ Save the corresponding coupling artifact.
    end if
  end for
  ▷ Compute EOT_i and update output coupling artifact.
  if $\mathsf{EOT}_i \neq \mathsf{Lookahead}_i(nt_i, EIT_i, t_{in_i})$ **then**
    $\mathsf{EOT}_i \leftarrow \mathsf{Lookahead}_i(nt_i, EIT_i, t_{in_i})$
    $\forall (k,j) \in \mathsf{OUT}_i : \mathcal{C}_j^i$.setLinkTime($\mathsf{EOT}_i$)
  end if
  ▷ Find the next secured (internal or external) event.
  if $(nt_i \leq t_{in_i})$ **and** $(nt_i \leq \mathsf{EIT}_i)$ **and** $(nt_i \leq \mathsf{Z})$ **then** ▷ If the event is internal.
    $\mathcal{I}_i$.processInternalEvent(nt_i) ▷ Process the event.
    for all $(k,j) \in \mathsf{OUT}_i$ **do** ▷ Send the resulting external output event.
      $eout_i^k \leftarrow \mathcal{I}_i$.getOutputEvent($y_i^k$)
      if $eout_i^k \neq \varnothing$ **then**
        $\mathcal{C}_j^i$.post($eout_i^k$, nt_i)
      end if
    end for
    $nt_i \leftarrow \mathcal{I}_i$.getNextInternalEventTime()
  else if $(t_{in_i} < nt_i)$ **and** $(t_{in_i} \leq \mathsf{EIT}_i)$ **and** $(t_{in_i} \leq \mathsf{Z})$ **then** ▷ If the event is external.
    $\mathcal{I}_i.processExternalEvent(ein_i, t_{in_i}, x_i^p)$ ▷ Process the event.
    $\mathcal{C}_i^c$.removeEarliestEvent(p)
    $nt_i \leftarrow \mathcal{I}_i$.getNextInternalEventTime()
  end if
end while

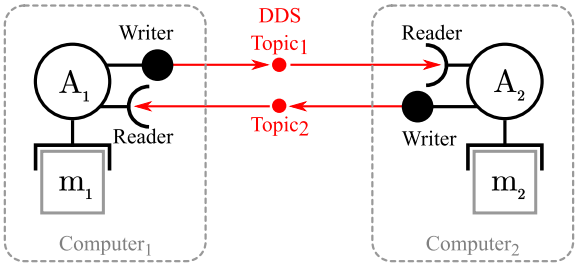


Figure 4: Distribution of a MECSYCO co-simulation.

to split the co-simulation. DDS being based on the publish-subscribe communication pattern, writer coupling artifacts play the role of publishers while reader coupling artifacts act as subscribers. Each writer coupling artifact sends data to its reader coupling artifact using a dedicated DDS topic (see Figure 4).

The UML diagram of Figure 5 shows how we implement the MECSYCO concepts following an object oriented programming. This implementation is in keeping with our multi-agent paradigm as each MECSYCO concept corresponds to a class of object, and each autonomous m-agent corresponds to a thread. We retain then the advantages of our paradigm: the software architecture is composed of a set of modular software bricks, enabling decentralized and parallel simulations.

In the following section, we detail how we wrap the FMI standard in DEVS using the hybrid M&S capacity of DEV&DESS.

6 DEVS Wrapping of the FMI Standard

As with any tools in MECSYCO, integrating an FMU requires to connect it to the co-simulation with a model artifact. This one exposes a DEVS view of the FMU, and makes the FMU handle discrete-events. To define such a model artifact, we can rely on the DEV&DESS formalism as it can be embedded into DEVS, and as it offers a sound framework for describing hybrid systems.

As defined by Zeigler [22], the DEVS version of a DEV&DESS model is composed of three components, each of them formalized as a DEVS atomic model. With this structure, a DEV&DESS model can be incorporated into a larger DEVS schema as a coupled model. Consequently, the DEV&DESS model can be simulated using the DEVS simulation protocol. The three components composing the model are:

- **A continuous component** describing the evolution of the continuous part of the system according to continuous inputs, and producing continuous outputs.
- **An event-detection function** determining when state-events occur, based on the continuous states of the model (i.e. the FMU state in our case).
- **A discrete-event component** describing the evolution of the discrete part of the system. This component describes the behavior of the model in the discrete-world, that is to say how it schedules internal events, how it produces and reacts to discrete inputs (i.e. external events), and what are the impacts of state-events. Potentially, for each of these events, the event-based component can change the whole DEV&DESS states, meaning (1) its own state, (2) the continuous component state (creating a discontinuity in the state trajectory) and (3) the event detection function.

As two versions of FMI exist, we propose two strategies to wrap FMUs in DEVS using DEV&DESS. The main difference between these strategies, which are detailed below, is the location of the continuous system solver: it is embedded into the FMU with FMI-CS, whereas it is implemented in the wrapper with FMI-ME. Each of these wrappers have pros and cons making them complementary.

- **FMI-ME** proposes primitives able to handle hybrid models. Moretheless, as stated in Section 3.2, an FMU for model-exchange (FMU-ME) needs to be associated with a solver to be simulated. Then, our DEV&DESS wrapper plays the role of an hybrid solver for this FMU-ME. In order to manage the continuous state simulation, the original Zeigler's DEVS version of DEV&DESS relies on a quantized integrator approach. The rationale behind this choice is that, quantized integrators have a discrete-event behavior as they quantize the state space instead of discretizing the time dimension. Thus, a quantized integrator naturally bridges the gap between the continuous and the discrete-event worlds [26]: its working principle is already based on the integration of inputs events and on the detection of state-events[23] (i.e. localizing when the state trajectory crosses a given threshold). As a result, it makes perfectly sense to keep this choice and to implement a quantized integrator in our wrapper. More precisely, we choose the QSS approach[23] (mainly developed by Kofman) as it offers some of the most advanced mathematical solutions for solving equation-based systems, while exhibiting striking simulation performances under some conditions. We currently have implemented QSS1 [48] (i.e. first order numerical method) and QSS2 [25] (i.e. second order numerical method) solvers for FMU.
- **FMI-CS** embeds a solver but does not yet include the primitives required for managing discrete-event behaviors [49, 32, 33] (e.g. the date of the

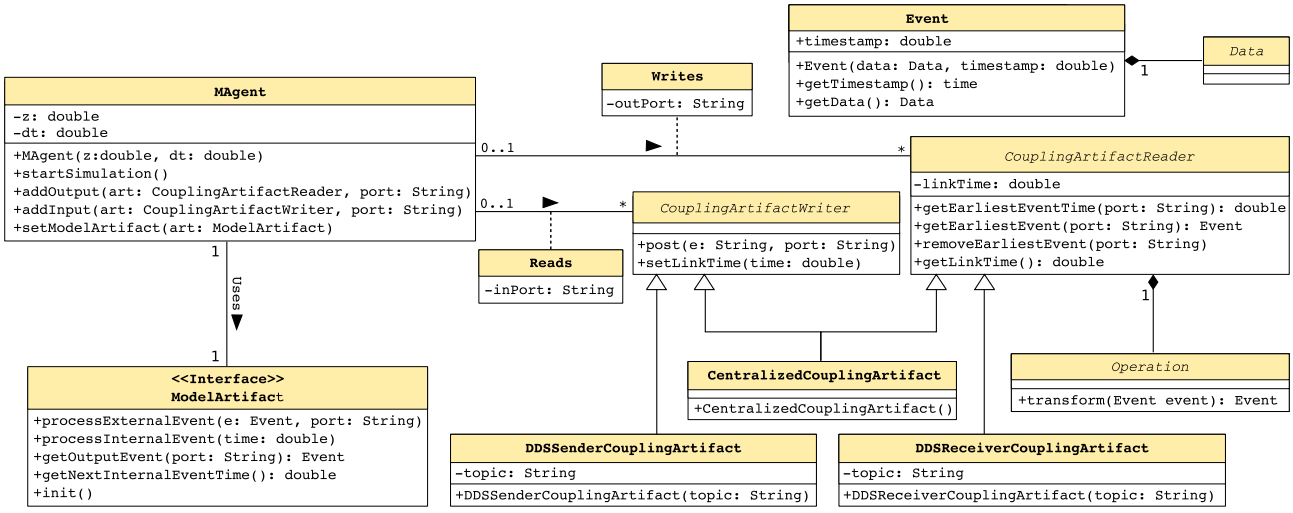


Figure 5: UML description of the MECSYCO software architecture.

next scheduled time-event can not be obtained from a FMU). Therefore, we consider that FMI-CS only specifies the continuous behavior of the system. We need then to specify its discrete behavior (i.e. the equivalent of the DEV&DESS event-detection function and the discrete-event component) within our wrapper. Additionally, specifying the discrete-event behavior outside the FMU enables a more flexible wrapping: different discrete-event behaviors can be associated with a single FMU depending on the co-simulation context (e.g. the discrete-event component can produce a discrete output signal by regularly sampling the continuous output of an FMU, or send events when the continuous output signal of the same FMU reaches a given threshold). Besides, when wrapping an FMU-CS in DEVS, we have to take account of an additional constraint: **the FMU is exported with its solver, and this solver can not belong to the QSS family** because, as stated before, FMI-CS can not handle such a discrete-event behavior. In consequence, **we can not use a QSS solver anymore here, and so we need to adapt the original DEVS version of DEV&DESS in this model artifact.**

The two next sections detail our wrappers and their assessments.

7 Wrapping of FMU for Model-Exchange

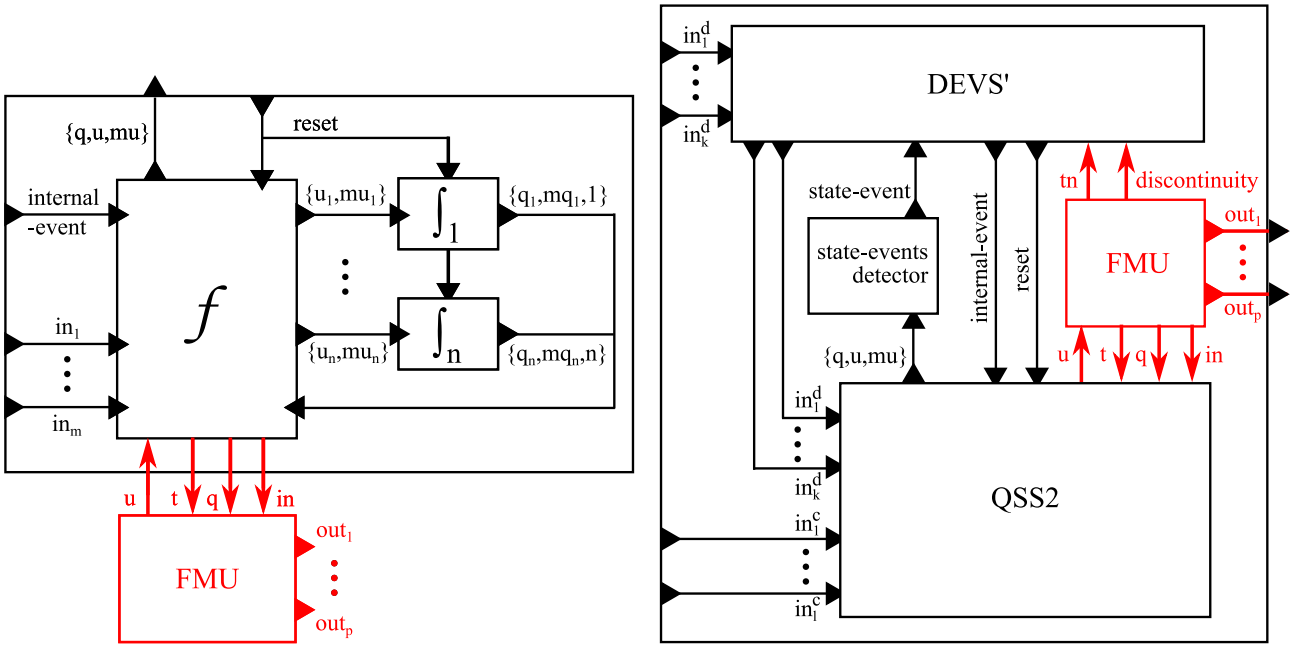
Figure 6a shows the architecture of our QSS2 solver for FMU. This architecture mainly follows the one defined by Kofman, but also has slight differences because two criteria were not handled by the original QSS specifications: (1) due to the FMU nature, the model is clearly separated from its solver, and (2) discrete-events may cause discontinuities in the continuous state trajectory. In the following sections we highlight these differences. First, in order to be understandable by non-specialist, we give an overview of the QSS working principle (Section 7.1). Then, we describe how our QSS solver works (Section 7.2) and how it interacts with the other components of DEV&DESS: the state-event detector (Section 7.3) and the discrete-event behavior component (Section 7.4). This whole structure of the wrapper is detailed in Figure 6b and corresponds to a DEVS coupled model, managed by a classic DEVS coordinator (not detailed here for sake of concision). This coordinator is directly controlled by the API of the MECSYCO wrapper. Finally, Section 7.5 details the assessment of the wrapper.

7.1 The QSS solver strategy

In order to explain the QSS method and to highlight its originality, we compare the behaviors of a first order quantized integrator (i.e. QSS1) and of a classical first order time-discretization integrator (i.e. Euler):

- Given the state x of a continuous system at a current time t_i, an **Euler solver** considers the first derivatives to remains constant during a time Δt. Based on this assumption, the solver infers the state of the system at the time $t_j = t_i + \Delta t$. Hence, the solver has a discrete time-stepped behavior with a time-step equals to Δt (see Figure 7a).

- Given the state x of a continuous system at a current time t_i, a **QSS1 solver** considers the first derivatives to remain constant until the system trajectory reach one of the thresholds $x \pm \Delta x$. Based on this assumption, the solver infers the



(a) QSS2 solver for FMU-ME.

(b) global view

Figure 6: bloc diagram view of the DEVS wrapper for FMU-ME.

time t_j ($t_i < t_j$) when the system state and derivatives needs to be updated. Hence, the solver has a discrete-event behavior (which can be described by a DEVS model) where events correspond to continuous state updates (see Figure 7b).

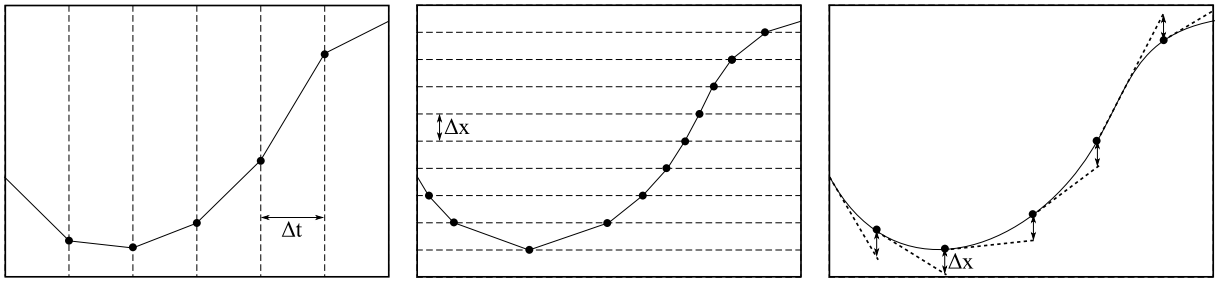
QSS2 uses a strategy similar to QSS1, but it performs a second order approximation. Hence, it considers that the **second** derivatives remains constant between events. Then, between events the system trajectory is approximated by a parabolic trajectory. An event now occurs when this parabolic trajectory differs from a linear one (which may correspond to a first order approximation of the system trajectory) of a quantity Δx (see Figure 7c).

7.2 Continuous Behavior Simulation with QSS

In the original QSS specifications, the solver interacts with two clearly separated function blocks which respectively define the output and the input behaviors of the model. In our wrapper, these blocks are directly embedded inside the FMU. Therefore, the outputs (both discrete and continuous) of the solver correspond to the FMU ones. The output ports of our wrapped coupled model are directly linked to the FMU ones. However, according to the standard, the FMU discrete output ports produce piecewise-continuous signals –i.e. these signals are always present no matter the time instant [33]. In order to generate discrete-event output signals (i.e. signals that are present only at some instants in time) for these discrete ports, we propose an optional mode in our wrapper which filters the output of the FMU in order to generate signals (i.e. external events) only at the moments of the time-events and/or the state-events.

According to the QSS approach, each variable x_i of the FMU continuous state vector is associated with a DEVS quantized integrator $\int_i$. Each integrator $\int_i$ takes in input the first and second derivatives of x_i respectively noted u_i and mu_i, and produces in output the new values and slopes of x_i, respectively noted q_i and mq_i. These integrators numerically solve the equation in an asynchronous way. A DEVS atomic model f is in charge of computing the derivative slopes, handling the inputs of the equation-based system -therefore, the model has a set $\{in_1..in_m\}$ of input ports, corresponding to the FMU ones- and interacting with the integrators. In the original QSS specifications, the equation-based system is directly embedded into f. This is not feasible in our case because the system is already embedded in an FMU. As a consequence, our solver f also manages the interaction with the FMU in the following way:

- When it has to update the FMU continuous state (e.g. when it receives new values and slopes for a continuous state variable, from an integrator), f first switches the FMU into the continuous mode (using `fmi2EnterContinuousTimeMode`) if it was not already, and call the `fmi2SetContinuousStates` function.



(a) Euler. (b) QSS1 (c) QSS2

Figure 7: Comparison of different solvers strategies

- When it has to update the value of an input variable of the FMU (i.e. when it receives input events through in_i ports), f first checks the variability of the variable into the XML description file. Depending if this variability is continuous or discrete, f calls the `fmi2EnterContinuousTimeMode` or `fmi2EnterEventMode` function in order to set the FMU in the appropriate mode (if it was not already). Then, f checks the input variable type in the XML file, and updates its value in the FMU using `fmi2SetReal/Integer/Boolean/String` function. If the updated variable is discrete, f asks (several times if required by the FMU) the FMU to re-evaluate its discrete state using `fmi2NewDiscreteStates`.

- When it receives any event at its input ports (e.g. from the integrators or at a in_i port), f updates the clock of the FMU to the timestamp of the events using the `fmi2SetTime` function.

- When it has to get the derivative u_i (e.g. in order to compute its slope mu_i and to forward these two values to $\int_i$), f uses the `fmi2GetDerivatives` function of the FMU.

As shown in Figure 6b the solver interacts with two atomic models in order to simulate the discrete behavior of the FMU. These models correspond to the ones defined by Zeigler in the DEVS version of DEV&DESS.

7.3 State-Event Detector

The **state-event detector** atomic model is in charge of the accurate localization of state-events during the simulation of the continuous equations. In order to take advantages of the QSS approach for detecting state-events, we make the hypothesis that the state-event thresholds of the FMU are *a priori* known (either because this information can be obtained from the model designer or from the XML description file). In the original hybrid QSS specifications[26], Kofman suggests two ways of feeding the state-event detector from the QSS solver:

1. It can receive the variable values q and derivatives u and mu. This way, as stated by Kofman, the detector only *"has to find the roots of a second degree polynomial"*[26] in order to find the time of the next state-event (in the absence of new state and derivative updates received from the solver). Then, the detector schedules an internal event at the time of this state-event in order to produce an output, notifying the occurrence of the event.

2. Or it can only receive the derivatives u and their slopes mu directly from the output ports of f. In this case, in addition to find the time of the next state-event and schedule the resulting internal event, the detector has to integrate (in parallel of the system resolution) the variables concerned with the thresholds.

Kofman opts for the second option because it does not imply any modification of the QSS solver. However, the drawback of this option is that the detector can not be aware of the discontinuities in the continuous state trajectory caused by discrete-events processing (time, state or external). This is why we choose the first option in our wrapper: the model f forwards immediately to the detector all the updates of the continuous states vector q and its derivatives u and mu, through a dedicated output port.

7.4 Discrete-event Behavior Simulator

The **DEVS** atomic model is in charge of managing the occurrences of discrete events (state, time and external). After each modification of the discrete state of the FMU –i.e. after each external/time/state-event processing in the FMU-, this component (1) retrieves the time tn of the next time-event scheduled in the FMU, and (2) checks if the event processing has created a discontinuity in the continuous state trajectory (by checking the information returned by the last call of the FMU `fmi2NewDiscreteStates` function). The DEVS component schedules an internal event at each tn. It also receives notifications of state-event occurrences from the detector. Moreover, all discrete in-

puts of the FMU are first sent to the DEVS component before being immediately forwarded to the QSS solver. This enables the DEVS component to be aware of discrete-input occurrences, and so to interact with the FMU (i.e. to update tn and check discontinuities) after the discrete input was processed by the solver. Therefore, as shown in Figure 6b, we distinguish in the QSS solver interface between:

- The set $\{in_1^c, ..., in_k^c\}$ of input ports, corresponding to the continuous inputs of the FMU. These ports are directly connected to the input ports of the wrapper. This way, the solver can directly receive continuous inputs of the FMU from the other simulation tools of the co-simulation.
- The set $\{in_1^d, ..., in_l^d\}$ of input ports, corresponding to the discrete inputs of the FMU. These ports are duplicated in the DEVS component interface.

As soon as it computes a time event or it receives a state-event notification, the DEVS component sends an internal event notification to the QSS solver through a dedicated port. The solver processes this notification in the same way it does with discrete inputs: it sets the FMU to the discrete mode and asks the FMU to re-evaluate its discrete state, thus causing the time/state-event to be processed. The only difference is that, as no discrete input of the FMU corresponds to this internal event notification port, the solver does not change any input variable of the FMU. Finally, as soon as the DEVS component detects a discontinuity in the continuous state trajectory, it sends immediately a reset event to the QSS solver through a dedicated port. According to Zeigler's DEV&DESS specifications, this event resets both the quantized integrators and the f model state, enabling the QSS solver to handle the discontinuity.

7.5 Implementation and Assessment

We have implemented this wrapper in the Java version of MECSYCO. In order to interact with the FMU, we rely on JavaFMI[50]. As this library only covers FMU-CS, we proposed an extension to interact with FMU-ME. We check the behavior of our wrapper by reproducing two QSS2 use cases proposed by Kofman[26]. The first one corresponds to an DC-AC inverter circuit equipped with switches controlled by discrete-inputs, which are sent according to a Pulse Width Modulation (PWM) strategy. The second example corresponds to a ball bouncing downstairs, with state-events occurring twice each bounce (one when the ball hits the ground and one when it leaves it). Note that, with this example, state-event occurrences depend on two continuous state-variables: when both x and y positions match the stairs location (i.e. when $y = floor(h + 1 - x)$). We translated Kofman's models into Modelica language (see Figures 8a and 9a) and exported them in FMUs for model-exchange, using OpenModelica. We found (visually) similar simulation results (see Figures 8b and 9b) and performances (i.e. a similar number of internal events) with our solver and with the Kofman one.

As these two models do not include discontinuities in the continuous state trajectory, we also propose another use case to test this aspect with our solver (see Figure 10a). This use case corresponds to the simulation of a barrel-filler factory inspired by the one proposed by Praehofer[13]. In this factory, we consider a queue of barrels waiting to be filled, on a conveyor. The factory fills only one barrel at a time. As soon as the water reaches a given level in the barrel, the barrel is carried away by the conveyor, and the filling process starts again for the next empty barrel. A tank stores the water to fill the barrels. The flow rate of water filling the barrel decreases with the level of water in the tank. A valve controls the flow of water between the tank and the barrel. The valve can only be in two states "open" (water goes from the tank to the barrel) or "close" (the filling process is stopped). The continuous dynamics of the model corresponds to the level of water in the current barrel and in the tank. The model receives discrete inputs controlling the valve. State-events correspond to the moment were the current barrel is full. At this point, the level of water in the current barrel is reset, to represent the barrel switching. The model produces a discrete output signal corresponding to a regular sampling of the level of water in the barrel. This signal can be for instance sent to a controller for monitoring the filling process. We found (visually) similar results when simulating this model with our QSS2 solver (see Figure 10b) and with OpenModelica solvers.

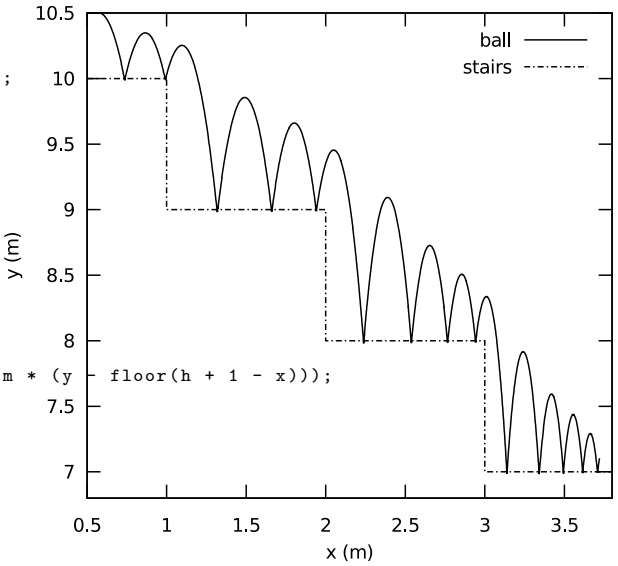
8 Wrapping of FMU for Co-Simulation

As stated in Section 6, we need the three components of DEV&DESS to integrate an FMU into DEVS. An FMU-CS provides the continuous behavior and we need to define the two remaining components (i.e. the state-events detector and the discrete-behavior component) in the wrapper. These components are dependent of the wrapping context:

- The discrete-behavior component has to specify the behavior of the FMU in the discrete world. This component corresponds to a DEVS atomic model able to interact with the FMU component. For example, this component can sample a continuous output of the FMU by regularly scheduling internal events, and producing external output events, according to the current values of the FMU variables, using `fmi2GetReal`.

```
model BouncingBall
  output Real x(start = 0.575) "horizontal position (m)";
  output Real y(start = 10.5) "vertical position (m)";
  output Real vx(start = 0.5) "horizontal speed (m/s)";
  output Real vy(start = 0) "vertival speed (m/s)";
  discrete Integer sw(start = 0) "discrete position";
  parameter Real k = 100000;
  parameter Real m = 1;
  parameter Real b = 30;
  parameter Real ba = 0.1;
  parameter Real g = 9.80665 "gravity (m/s^2)";
  parameter Real h = 10 "first step height (m)";
equation
  vx = der(x);
  vy = der(y);
  der(vy) = (-g) - ba * vy / m - sw * (b * vy / m + k / m * (y - floor(h + 1 - x)));
  der(vx) = -ba / m * vx;
  when y <= floor(h + 1 - x) and pre(sw) == 0 then
    sw = 1;
  elsewhen y >= floor(h + 1 - x) then
    sw = 0;
  end when;
end BouncingBall;
```

(a) Modelica code of the model.

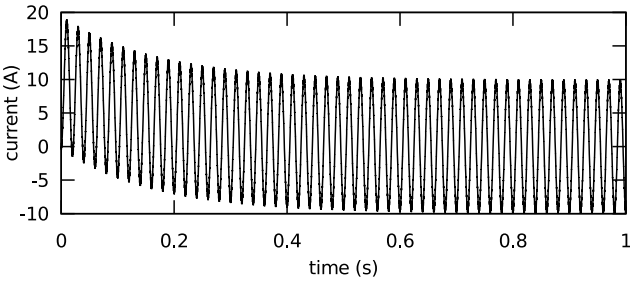


(b) MECSYCO simulation results.

Figure 8: Simulation of the bouncing ball system.

```
model InverterCircuit
  parameter Real R = 0.6 "Resistance (ohm)";
  parameter Real L = 0.1 "Inductance (H)";
  parameter Real Vin = 300 "Input Voltage (V)";
  Real iL(start = 0) "Current (A)";
  discrete input Integer sw(start = -1) "switch";
equation
  der(iL) = (-R * iL + sw * Vin) / L;
end InverterCircuit;
```

(a) Modelica code of the model

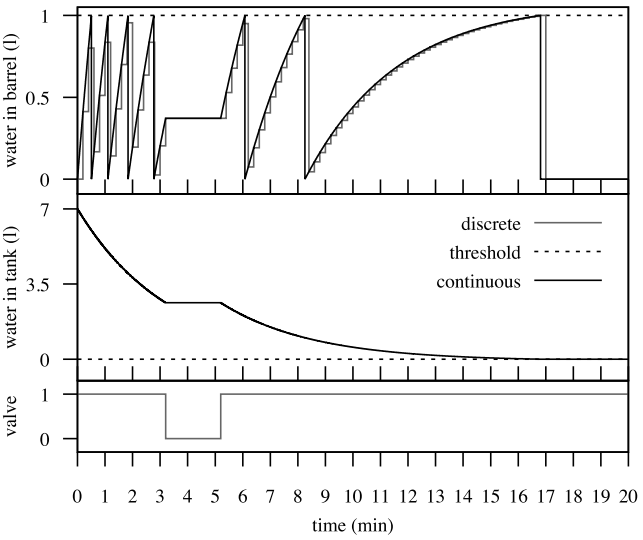


(b) MECSYCO simulation results.

Figure 9: Simulation of the DC-AC inverter circuit.

```
model barrel
  discrete input Boolean valve(start = true);
  parameter Real qmax = 7 "initial tank water (L)";
  Real q(start = qmax) "water in the tank (L)";
  Real flow(start = 0) "flow of water";
  parameter Real gain = 0.3;
  parameter Real value = 0.025;
  output Real x(start = 0) "water in the barrel";
  parameter Real xmax = 1 "wanted barrel water(L)";
  parameter Real f = 10 "sampling frequency";
  discrete output Real y(start = 0) "output sampling";
  equation
    flow = if valve and q > 0 then q*gain+value else 0;
    der(q) = -flow;
    der(x) = flow;
    when x >= xmax then
      reinit(x, 0);
    end when;
    when sample(0, 1 / f) then
      y = x;
    end when;
end barrel;
```

(a) Modelica code of the model.



(b) MECSYCO simulation results.

Figure 10: Simulation of the barrel-filler factory.

- The state-events detector has to specify the condition of occurrence of state-events, according to the FMU state. This detector corresponds to a boolean function $S \rightarrow \{true, false\}$ with S the set of the FMU states. For example, this function should return true (i.e. a state-event occurs) only when a variable of the FMU is superior or equals to a given value.

In the following, we detail how we implement the main DEVS primitives into the wrapper.

8.1 Time of the Next Internal Event

In our DEVS wrapper for FMU-CS[51], we rely on the FMI specifications to simulate the continuous output of the component: we consider that the FMU produces outputs at a sequence of pre-defined communication points. From our DEVS point of view, these communication points are seen as internal events producing external output events. In the same way, from our DEVS point of view, we see updates of the continuous input values received by the FMU as input events.

According to the DEVS semantics, the `getNextInternalEventTime()` function must return the date of the earliest scheduled internal event in the model. In the DEV&DESS context, this date corresponds to the minimum between:

- the date of the next internal event scheduled in the discrete-event component;
- the date of the next communication point of the FMU,
- and the date of the next state-event.

Getting the first two dates is trivial as they are *a priori* known. Things get more complex for the state-events: because of the numerical resolution of the equational model, state-events can only be detected *after* each integration step of the FMU, and their localization in time can only be approximated.

In order to get the date of the next state-event, we need to perform an exploration with the FMU to see if a state-event will occur before its next communication point. As a consequence, the component will always be "in the future" compared to the current simulation time. According to the DEVS semantics, the `getNextInternalEventTime()` function must not change the state of the model. Indeed, it is imperative to be able to come back to the previous state of the FMU, corresponding to the only legitimate state from the simulation point of view. The rollback capability of the FMU assures this feature as long as no new integration step is performed.

When a state event is detected during an exploration, we perform a bisectional search [16, 52] in order to localize the state-event as precisely as possible in the time. This search is formalized by the Algorithm 2 which, given the initial integration step ΔT and a number of iterations m (formalizing the search precision), positions the FMU as close as possible to the state-event occurrence. The algorithm basically progresses by a succession of integration steps whose duration δt is adapted according to state-event occurrences, and following a dichotomous strategy. As the original state must always be accessible, and as only one integration step can be canceled at a time, the algorithm always goes back to the legitimate state before performing a new integration step.

Algorithm 2 Bisectional search for state-event localization.

INPUT: $\Delta T \in \mathbb{R}_0^+, m \in \mathbb{N}_0^+$
  $\delta t \leftarrow 0$
  $\Delta t \leftarrow \Delta T$
  for 1 **to** m **do**
    `fmu2RollBack()`
    $\Delta t \leftarrow \Delta t/2$
    `fmi2DoStep`$(\delta t + \Delta t)$
    if ¬`stateEventOccurence()` **then**
      $\delta t \leftarrow \delta t + \Delta t$
    end if
  end for

8.2 Events Processing

According to the DEV&DESS semantics, when an event (internal, external or state-event) occurs at simulated time$t$, the equational component describes the continuous evolution of the system until t, and the event is processed by the discrete-event component. This behavior is translated in our model artifact as follows.

When the `processExternalEvent`(e_{in_i}, t, x_i^k) function is called to report the occurrence of an external input event e_{in_i} into the x_i^k input port, the first step consists in rolling back the FMU to its previous state (using the `fmu2RollBack()` method). This one corresponds to, as stated in the previous section, the only legitimate state from the simulation point of view. Then, the FMU performs an integration step until t (using the `fmi2DoStep` method) in order to reach the point where the event occurs. Finally, when x_i^k is a continuous port, the FMU is parametrized accordingly (using the `fmi2SetReal/Integer/Boolean/String` method). If x_i^k is a discrete port, the external transition function of the discrete-event component is triggered in order to process e_{in_i}.

Similarly, when the `processInternalEvent`(t) function is called to process the next internal event, the FMU is rolled back to its previous state and an integration step is performed until t (using the `fmi2DoStep` method). On the one hand, if the

next internal event corresponds to a communication point of the FMU, then the model artifact retrieves the continuous output ports values (using the `fmi2GetReal/Integer/Boolean/String` method), and produces the external output events accordingly. On the other hand, if the next internal event corresponds to a state-event or the next internal event of the discrete-event component, then the internal transition function of this latter is called, which could produce external output events.

9 Discussion

We have presented in Section 5 the whole specification of the MECSYCO middleware dedicated to the co-simulation of CPS using a DEVS wrapping strategy. MECSYCO relies on the formal guarantees offered by DEVS and on the practical guidelines offered by the numerous integrative works around DEVS in the literature to rigorously integrate models written in different formalisms.

As a consequence, MECSYCO inherits the DEVS limitations –i.e. if a M&S tool uses a formalism that can not be integrated in DEVS, then the tool can not be integrated in MECSYCO. We also stress that in order to ease the wrapping of tools, we use the classical version of the DEVS formalism (i.e. we do not consider the Parallel DEVS formalism here): this prevents to make assumptions on how an existing model reacts to simultaneous event. As a consequence, simultaneous events may not be taken into account in a reproducible way (i.e. their processing order may vary, which may impact the results).

The MECSYCO co-simulations are coordinated in a decentralized and conservative way with a parallel execution thanks to the Chandy-Misra-Bryant algorithm. Having a conservative algorithm ease the integration of tools by not requiring the roll-back features. However, it is worth noting that depending on the co-simulation characteristics, parallel optimistic or sequential co-simulation execution can be more efficient.

Even though the models can be interfaced from a software and formal perspective with MECSYCO, there is no guarantee that they can be composed in a *meaningful* way (i.e. resulting in a co-simulation semantically valid) [53]. Hence, every MECSYCO co-simulations and every DEVS wrappers must be carefully validated and verified in order to bring exploitable results. The verification of the MECSYCO implementation was done empirically through several use cases (notably in collaboration with EDF which was able to check the results against a real system [44], and when studying green cloud computing [43]), but no formal verification were performed.

In Section 6, we showed how FMU components can be wrapped into DEVS. The integration of the FMI standard gives a way (in term of software interface) to integrate at once continuous models developed using various tools (e.g. Dymola, MATLAB/Simulink). We stress that, according to our wrapping strategy, we do not directly map an FMU into a DEVS model but rather provide additional mechanisms (using QSS and DEV&DESS) in order to control an FMU like a DEVS simulator. We stress that as we base this wrapping directly on the DEVS protocol, this work is not limited to the MECSYCO platform, but can be implemented in any DEVS-based platform. We proposed two wrappers in order to integrate two complementary kinds of FMU proposed by FMI (namely Model-Exchange and Co-simulation).

With our DEVS wrapping of FMU-ME, we define a hybrid QSS solver tailored to the FMI standard. At this point, other QSS versions of the literature [54, 55] which can simulate hybrid Modelica models deserve to be cited. Our solver can also simulate models written in Modelica, as soon as they are exported into an FMU-ME. However, the originality is that our QSS solver can also solves models written in any of the numerous software compliant with FMI for model- exchange (e.g. MATLAB/Simulink). Yet, it is important to note that FMI prevents us to fully exploit all the performances of the QSS method. Indeed, FMI does not allow to decompose the continuous system in order to individually update the continuous state vector elements. As a result, the QSS algorithm (i.e. the integrators) can not solve the system asynchronously. Hence, this limit of the FMI standard could make QSS inefficient for solving large ODE systems. Moreover, as so far we only provided a QSS2 solver which is only of order 2, we are still strongly limited to simple non-stiff equation-based models. In order to be able to simulate more realistic use cases, we plan in future works to implement other QSS methods such as QSS3 [56] (of order 3), and LIQSS2[57] (of order 2, but for stiff systems).

In the case of FMU-CS, we would like to underline the fact that, whereas our wrapping of FMU for model exchange is adapted both for FMI 1.0 and 2.0 versions, our wrapping of FMU-CS is only adapted for FMI 2.0. This is because we needed the rollback capacity of the FMU which is only available in the latest version of the standard. Besides, this rollback capacity is only optional in FMI 2.0. Consequently, our DEVS wrapper is unable to handle an FMU-CS which does not implement this feature. It is also worth noting that we made the assumption that FMUs for co-simulation always accept the desired integration step. This assumption is not trivial because the FMI standard does not prescribe an FMU to reject the required integration step to to prematurely stop the numerical integration of the system [32]. Depending on the solver exported within the FMU, it could happen for instance when the estimated error becomes too large or when the solver

has a fixed step size incompatible with the required integration step. Thus, our wrapper may not be compliant with all FMU-CS behaviors. As a result, when exporting a model into an FMU-CS for a MECSYCO wrapping purpose, the solver must be carefully selected (when available). In particular, solvers with fixed step-size should be avoided here.

These two wrappers can be considered as complementary:

- FMU-ME wrapper can be used to integrate any hybrid system whose continuous behavior can be simulated by a QSS solver.
- FMU-CS wrapper can be used to integrate any purely continuous system which can be simulated with a solver compliant with FMI and the aforementioned assumptions. An ad-hoc discrete behavior can be specified in the wrapper if needed.

Please note that, if some continuous/hybrid model does not comply to any of these two wrappers, an ad-hoc wrapping of its tool may still be performed.

In the following section, we show the features of our solution, through a proof of concept of a smart heating M&S.

10 Use Case

Our use case is inspired by different works around smart-heating [55][58]. We want to simulate the evolution of the temperature and the power consumption of two buildings equipped with electric heaters. Using this simulation, we are interested in the design of a controller for limiting the consumption peaks duration in the building. To do so, this controller temporarily disables some heaters according to the information it receives on the building temperatures and power consumption. This controller interacts with the buildings system through an IP telecommunication network. Such a goal could lead to a typical iterative M&S process driven by the following series of questions:

1. What are total the power consumption and the temperatures evolution in the buildings, without a controller?
2. Does the controller actually achieve its goal, without considering delays and perturbations potentially induced by the telecommunication network?
3. What is the impact of the telecommunication network presence, on the controller performances?

This leads to three major steps in the M&S process.

In order to answer to the first question, we need to simulate the thermic system. We use three models. One describes the outside temperature evolution. The two others describe the power consumption and temperature evolution of each building, according to the outside temperature evolution. We perform the co-simulation of these three models by feeding the building models with the outside temperature trajectory.

In order to answer the second question, we build the model of the controller. We use this model twice (one for each building) in the co-simulation. Each controller model is fed with the outputs of its building model (i.e. room temperatures and heater power consumption). When needed, it produces the heaters switch off/on orders as output, sent to the building model as inputs.

In order to answer the last question, we add a model of the telecommunication network between the buildings and their controller. Outputs of the buildings models now first pass through the network model before arriving to the controller models. Reciprocally, the controller orders transit through the network model before delivery. The network model adds delays and perturbations (i.e. packet loss and noise) to the system.

This "toy" use case does not claim to be realistic. We keep the atomic models of the use case simple since we are here focused on demonstrating the following MECSYCO properties, rather than on presenting a credible use case:

- **Modularity:** The use case development follows an iterative M&S process. We first begin the co-simulation with the thermic model of the building. Then, we add step by step the models of the controller and the telecommunication network. We show that passing from one of these steps to another does not require to rebuild the co-simulation from scratch.
- **Software interoperability management:** Each model of the co-simulation is implemented in a different simulation software. The thermic model is defined in Modelica and exported into FMUs for model-exchange and co-simulation, the telecommunication model is defined using the NS-3 simulator, and the controller model is implemented in an ad-hoc way using the Java language. We show that MECSYCO properly handles exchanges of data between these heterogeneous software.
- **Multi-formalism integration:** The models of the co-simulation are defined in different formalisms. The thermic model is an hybrid model composed of differential and discrete equations. The telecommunication model is a discrete event model whereas the controller model is a discrete time-stepped model. We show that MECSYCO enables the rigorous integration of these heterogeneous models.

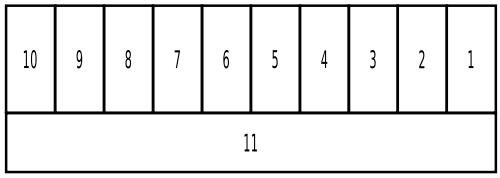


Figure 11: Architecture of the building

- **Multi-representation integration:** The models evolve at different temporal scales: seconds for the controller and the thermic models, and nanoseconds for the telecommunication network. We show that MECSYCO rigorously synchronizes these model executions during the co-simulation.
- **Distributed multi-platform execution:** We execute the co-simulation on two computers connected on a LAN. These two computers use different operating systems, and different implementations of MECSYCO. The telecommunication network model is executed on GNU/Linux Debian with the C++ version of MECSYCO, whereas the other models are executed on Microsoft Windows 10 with the Java version of MECSYCO.

In order to make this use case reproducible and to describe in a transparent way all its heterogeneity, we detail all models and their implementation in the following sections. Finally, in Section 10.4 we describe the different co-simulations made with these models, we discuss the simulation results, and we highlight the benefits offered by MECSYCO.

10.1 Thermic System Models

We create two kinds of models for the thermic system. The first one corresponds to the outside temperature trajectory. For sake of simplicity, this model generates a simple sinusoidal signal representing day/night temperature cycles. The second one corresponds to the temperature and power consumption evolutions of one building. As the two buildings are identical, we use this model twice.

Each building of the thermic system is composed of ten rooms linked by a corridor, following the Figure 11. Each room is influenced by the outdoor temperature, by the adjacent rooms, and contains an electric heater with an internal thermostat. This one turns on when the temperature inside the room falls under a minimal value and turns off when this temperature reaches a maximum value. For sake of simplicity, we assume here that all heaters have the same features (i.e. setpoint temperatures, powers and tolerances).

We use OpenModelica to define these models. The following list presents the interface of the building model used to interact with the other models of our use case.

Inputs:

- $blackout_i$ is a discrete boolean input. When set to true, the electric heater of the room i is shut down.
- T_{out} is the continuous outside temperature in K.

Outputs:

- R_iTemp is a discrete variable sampling the temperature of the room i. This signal is updated every *period* of time, and represents the information sent regularly by a thermometer to the controller.
- R_iPow is the instantaneous power consumption inside the room i. It is a discrete variable updated each time the heater starts and stops.

The building model is an hybrid system which combines continuous and discrete behavior. The simulation of this model requires to solve the differential equations system describing the temperatures evolution while taking account of discrete time and state-events. These discrete events correspond to the Modelica "when" statements. Each update of the discrete output ports R_iTemp corresponds to a time-event scheduled in advance by the model, for regularly sampling the continuous temperature evolution. On the contrary, a state-event occurs each time the temperature of a room reaches one of the two heaters thresholds –i.e. each time $T_i = T_{wanted} \pm \frac{bandwidth}{2}$. Considering the 11 rooms of the building, 22 state-event thresholds have then to be simultaneously monitored. Moreover, the continuous inputs of the outside temperature and the discrete inputs corresponding to the blackout orders of the controller have to be integrated during the simulation. More details about this model can be found in Appendix A.

10.2 Controller Model

The controller model is built in an ad-hoc way in Java following the DEVS functions of our model artifacts. As the two buildings do not interact together (i.e. no temperature exchanges occurs between the buildings), the controller can manage each building separately. Then, we define here the model of the controller for managing only one building. This model can be duplicated in order to control both buildings.

Recall that the goal of the controller is to limit power consumption peaks duration in the building. To do so, the controller temporary disables some heaters when the total power consumption of the building is equal or higher than a given threshold Pow_{max}. Hence we accept to lower the temperatures of some rooms beneath the setpoint, for a specific period of time. Nevertheless, in order to maintain a minimum of comfort in every room, the controller makes sure that the temperature

is above a given threshold $Temp_{min}$ in K (assumed to be lower than the temperature setpoint of the heaters).

The controller maintains a set of variables $Temp_i$ and Pow_i for saving respectively the last temperature and the last instantaneous power consumption values received from the sensors of each room i of the building. This controller is described by a discrete time-stepped model where each time-step corresponds to an evaluation point. Each Pow_i and $Temp_i$ variable can be updated by specific input ports of the model. The controller order to the heater of each room i corresponds to a boolean sent through a specific output port $blackout_i$. From our DEVS wrapping perspective, we consider each time-step as an internal event and each input/output as an external event. More details about this model can be found in Appendix B.

10.3 Telecommunication Network Model

The IP network is modeled with NS-3 [6], a popular discrete-event IP network simulator. NS-3 models can be wrapped into DEVS, as a coupled model composed of network components [37]. From the perspective of the IP network, each room corresponds to two network devices. A heater sends information about its power consumption to the controller, and receives commands from this latter, asking them to stop heating for a while. A thermometer regularly sends the current temperature of the room, to the controller too. More details about this model can be found in Appendix C.

10.4 Co-Simulations and Results

This section details the co-simulations and their results. The parameters used in the different co-simulations are provided in Table 3.

In order to answer to the first question, we export the thermic building model into an FMU-ME, to handle discrete-events. As said previously, we use two instances of this model, one for each building we want to simulate. We export the outside temperature model as an FMU-CS called *Out*.

According to our wrapping strategy, each building FMU is associated with an instance of our QSS solver. Each of these QSS solves the 11 differential equations of its models and monitors its 22 state-event thresholds. We set the quantization of all the integrators of the solvers to 0.0001. As shown in Figure 12b, we interconnected the wrapped models in MECSYCO in order to form the DEVS coupled model of Figure 12a.

The co-simulation is executed on a single computer using Windows 10 and the Java implementation of MECSYCO. We simulate one day of the system evolution.

The co-simulation results are shown in Figure 15a. For the sake of concision, these results only show the state trajectory of the first building. These results are similar to the ones obtained with OpenModelica, and perfectly match the expectation: state-events are handled at the right times (i.e. heaters start and stop just when the temperatures evolution reaches one of the two thresholds), and we can see the influences of the building symmetry with room 1 to 10 in the state trajectory (e.g. the rooms 1 and 10, or the rooms 5 and 6, which receive similar thermal influences, have similar trajectories).

In order to answer the second question, we add the two controller models (one for each building) to the co-simulation. According to the co-simulation parameters, the controller considers that consumption peaks occurs when the total power consumption of the building is higher than the power consumption of one active heater (i.e. when at least two heaters are active at the same time). We configure the models in order to have the controller evaluating the building states every minute. 30 seconds after each evaluation point (and its potential orders sent to the heaters), the controllers receive new information from the building sensors, and wait another 30 seconds until the next evaluation point. We connect the wrapped model in order to form the coupled model of Figure 13a.

The Figure 15b shows the simulation results for the first building. In this graph, grey areas represent periods of time during which the heaters should be shut down according to the controller model outputs. Again, the simulation results are in accordance with the expected model behaviors. Indeed, we can see that the controller model outputs are well integrated into the building model: when the controller sends the shut down orders, the heaters immediately stop working, and the corresponding room temperatures start decreasing according to the wall heat transfers. On the contrary, as soon as the controller sends starting orders to the heaters, the corresponding temperatures immediately start increasing and oscillate as expected between the two state-event thresholds.

In order to answer the last question, we add the telecommunication model to the co-simulation, as indicated by the Figure 14. As NS-3 works at a nanosecond timescale whereas the FMUs use a second time scale, we use transformation operations in the coupling artifacts between NS-3 and the FMUs (converting the timestamps of the exchanged events).

It is important to note that the models are compliant with different OS: FMU components we have generated are only compliant with Microsoft Windows, whereas the NS-3 model works on GNU/Linux. Moreover, we used different implementations of MECSYCO to wrap our models: the FMU components and the controller model are wrapped using the Java version whereas the NS-3 model is wrapped using the C++ version. As a

consequence, we have to distribute the co-simulation on two computers:

- The first one runs on Windows 10 and uses the Java version of MECSYCO to simulate the FMUs and the controller model.
- The second one runs on GNU/Linux Debian and uses the C++ version of MECSYCO to simulate the NS-3 model.

When we configure NS-3 for simulating a TCP protocol on the network without any error model, the simulation results are similar to the previous ones (shown on Figure 15b) –i.e. the network does not impact the system behavior. This is because, in this case, the network only introduces very small delays (on a second time scale) in the communications between the buildings and the controller. However, when we configure NS-3 with perturbations introduced in the simulated communications (i.e. packets losses or corruptions), the simulation results change as shown by Figures 15c and 15d. Perturbations are introduced by using in NS-3 an UDP protocol without checksums and an error model of one bit altered respectively every 10000 ones, then every 1000 ones. We can see that, as one can expect, the more noise we add in the network, the more different the system trajectory becomes. It is interesting to note that in the results shown by Figure 15d, the noise is so high that some controllers orders (for instance the shutdown order for the heater of room 1 at time 8370) do not even reach the building. Note that Figures 15c and 15d only display an example of simulation results, as the NS-3 error model introduces a stochastic process.

10.5 Synthesis

With this use case, we have shown that MECSYCO can rigorously integrate different kinds of heterogeneity. At each step of this use case, we introduced a new heterogeneity at the software, formalism and representation levels.

- The first step shows that MECSYCO handles the FMI standard (both co-simulation and model exchange), and hybrid dynamics (i.e. continuous evolution with state and time events).
- The second step shows that MECSYCO enables the interaction of continuous and time-stepped models, and properly manages the data exchanges between FMUs and ad-hoc simulators.
- The last step shows that the NS-3 discrete-event simulator can rigorously interact with FMUs and ad-hoc models in a distributed multi-platform architecture within MECSYCO.

Through this iterative proof of concept, we have shown that MECSYCO enables the modular M&S of a CPS. Indeed, it is important to note that, at each next co-simulation step, we only add and connect the new models to the previous co-simulation. Hence, we do not have to modify neither the models nor their MECSYCO wrappers: we only have to change the co-simulation structure (i.e. models interconnections and co-simulation distribution).

11 Conclusion

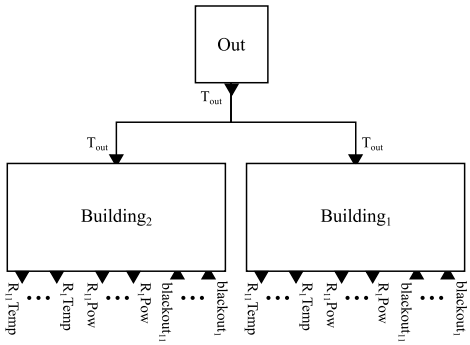
In this work, we gave the specifications of the MECSYCO middleware. MECSYCO tackles the numerous and difficult challenges of the CPS co-simulation. For this purpose, it relies on a DEVS wrapping strategy. The middleware performs the co-simulation in a parallel, decentralized and distributable fashion thanks to its modular multi-agent software architecture.

In this article, we illustrated how the DEVS literature and tools (namely the DEV&DESS hybrid formalism and the QSS solver strategy) can be used to rigorously integrate pre-existing equation-based tools into the MECSYCO discrete environment to perform hybrid co-simulations. In order to make this integration the more generic possible at the software level, we defined DEVS wrappers for the FMI standard. As a consequence, this DEVS wrapping of the FMI standard is reproducible in any DEVS-based platform.

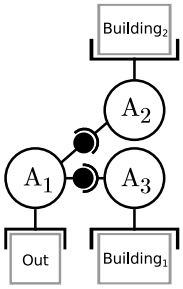
We developed a proof of concept of a smart-heating use-case, where we integrate and co-simulate non DEVS-centric M&S tools, namely OpenModelica and NS-3. We showed that our middleware is modular: there is no need to change the middleware specifications when a model is changed/added/removed in the co-simulation. Moreover, our middleware is fully specified from the concepts, till their implementation, making different implementations of MECSYCO interoperable.

Our approach is grounded on choices at different levels with resulting properties and limitations:

Using DEVS enables MECSYCO extensions based on DEVS literature. For example, dynamic structure and other synchronization algorithm (e.g. optimistic) could be implemented. The DEVS wrapping strategy offers two advantages: (1) pre-existing legacy M&S tools and their models can be re-used (2) we benefits from all the integrative works around DEVS in order to rigorously integrate tools using different formalisms. However, MECSYCO also inherits of DEVS limitations: if a formalism or a solver can not mapped (directly of indirectly) into DEVS, it can not be integrated in MECSYCO. On a more specific level, by relying on sequential DEVS instead of Parallel DEVS,

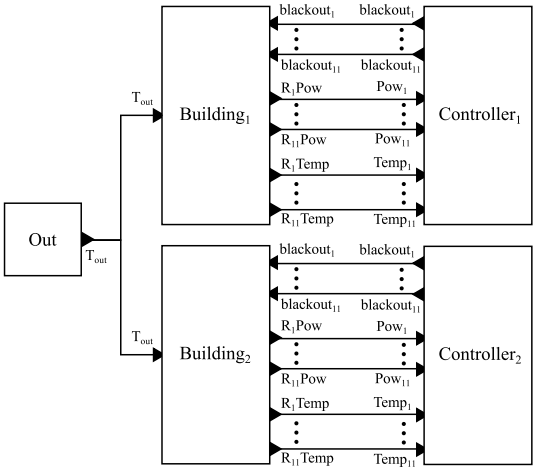


(a) Bloc diagram view of the DEVS model.

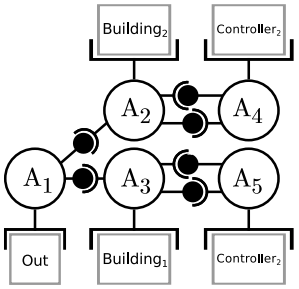


(b) MECSYCO view of the co-simulation.

Figure 12: Co-simulation of the building system without controller.

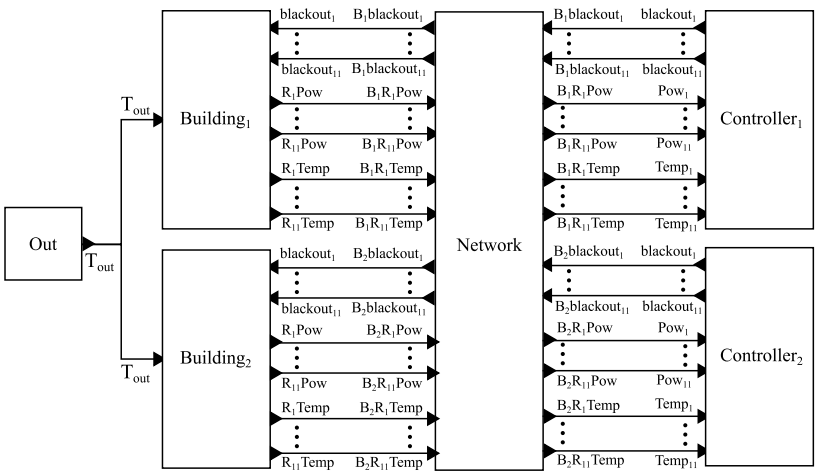


(a) Bloc diagram view of the DEVS model.

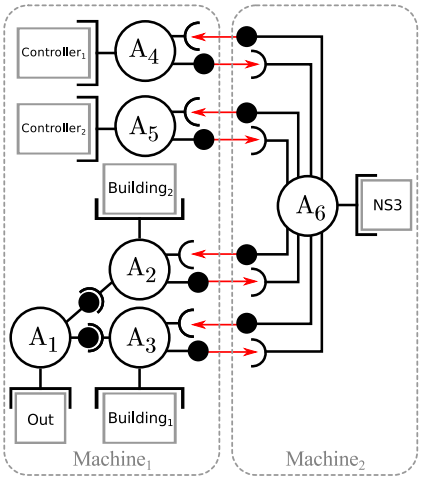


(b) MECSYCO view of the co-simulation.

Figure 13: Co-simulation of the building system with a controller but no network

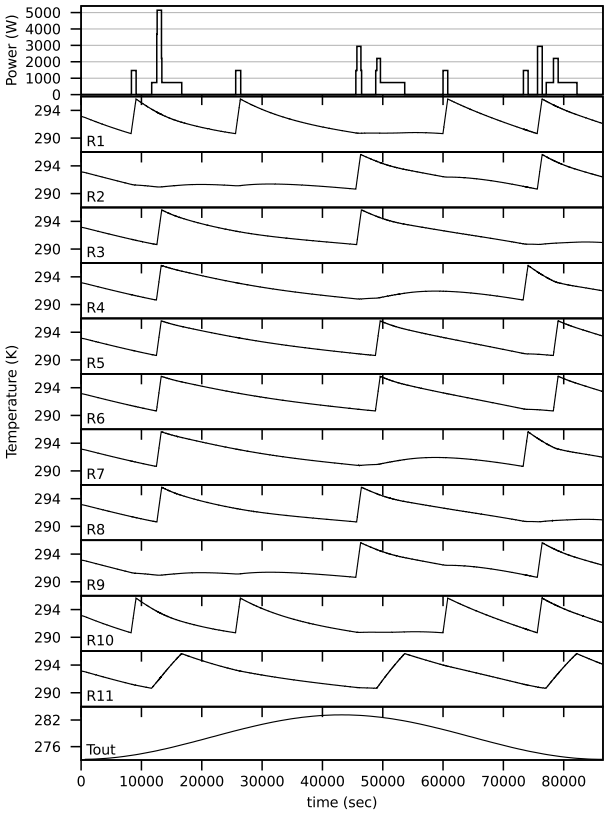


(a) Bloc diagram view of the DEVS model.

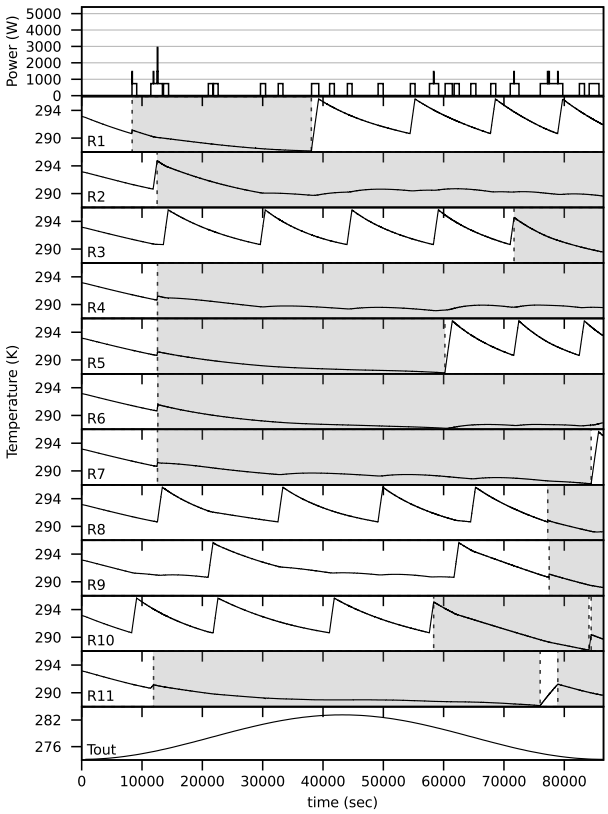


(b) MECSYCO view of the co-simulation

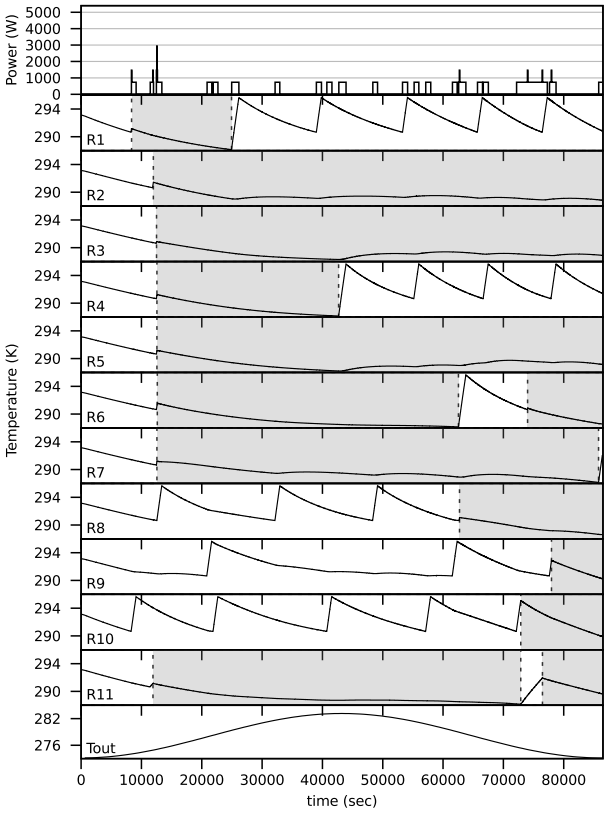
Figure 14: Co-simulation of the building system with a controller and a network.



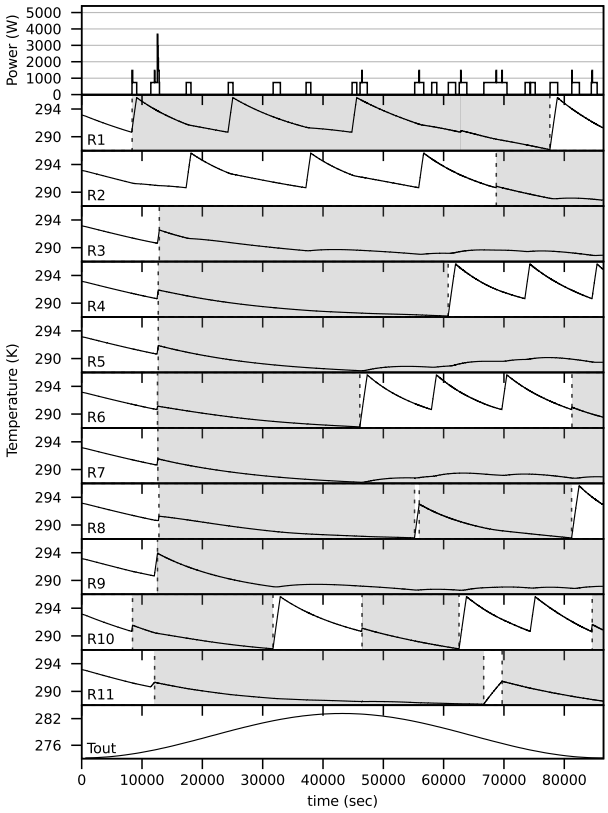
(a) Simulation results without a controller.



(b) Simulation results with a controller but no network.



(c) Example of simulation results with UDP and 1 bits altered every 10000 ones.



(d) Example of simulation results with UDP and 1 bits altered every 1000 ones.

Figure 15: MECSYCO co-simulation results of the building-controller system.

we ease the wrappers design but we may not take account of simultaneous events in a reproducible way.

Using a parallel conservative co-simulation algorithm simplifies the integration of tools because of limiting assumptions (e.g the roll-back features is not mandatory) but it forbids to take advantage of optimistic or sequential ones.

Using the FMI standard to integrate continuous systems, enables to have a more generic software compatibility: we can use all the tools compatible with FMI. However FMI is still not natively supported by all the equation-based tools (e.g. PowerFactory). As a consequence, ad-hoc wrappers must be developped for these tools. We proposed two complementary kinds of wrappers for FMI:

- In the case of FMI-ME, we have to code the solver in the wrapper. Thus, with this strategy, we can not reuse already implemented solvers. However, the advantage is that we can integrate and simulate hybrid models (e.g. with continuous AND discrete behaviors). In our case, we used QSS solver of order 1 or 2 to simulate the model. As we only provide a QSS2 solver we are limited to simple non stiff equation system even if other QSS solvers can be implemented in the platform. This approach may also be inefficient for large ODE system because FMI prevents us from fully exploit the QSS method.
- In the case of FMI-CS, we have to use the solver embedded in the FMU. This means that, when exporting a model as a FMU from a continuous tool, any solver compliant with the standard and our assumptions can be chosen. Thus, we are only limited by the integrative power of FMI and our assumptions. Nonetheless, the discrete behavior has to be rewritten in the wrapper. Also, note that we are limited to FMUs 2.0 which provide a roll-back capacity and must always accept the desired integration step.

In future works, we plan to propose extensions of our approach in order to have MECSYCO supporting the whole M&S process, from the definition of the experimental plan to the simulation results analysis. This includes the verification of the wrappers to guarantee the correctness of tools integration in the platform. We also plan to enhance the capacity of MECSYCO by implementing other DEVS co-simulation algorithms (e.g. optimistic or centralized). Finally, we would like to develop a Domain Specific Language approach within MECSYCO, to define co-simulations directly using the language of experts. Such an approach could indeed make MECSYCO accessible outside the M&S experts circle.

Acknowledgements

This work was partially funded by EDF R&D through the strategic project MS4SG.

References

[1] Rajkumar RR, Lee I, Sha L et al. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*. DAC '10, New York, NY, USA: ACM, 2010. pp. 731–736.

[2] Dahmann JS, Fujimoto RM and Weatherly RM. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*. 1997. pp. 142–149.

[3] Diallo SY, Herencia-Zapana H, Padilla JJ et al. Understanding interoperability. In *Proceedings of the 2011 Emerging M&S Applications in Industry and Academia Symposium*. EAIA '11, San Diego, CA, USA: SCS, 2011. pp. 84–91.

[4] Wilensky U. Netlogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999. URL http://ccl.northwestern.edu/netlogo/.

[5] Taillandier P, Vo DA, Amouroux E et al. GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In *Principles and Practice of Multi-Agent Systems*. Springer, 2012.

[6] Henderson TR, Roy S, Floyd S et al. NS-3 project goals. In *Proceeding of WNS2 '06*. 2006. p. 13.

[7] Varga A and Hornig R. An overview of the OMNeT++ simulation environment. In *Proceedings of ICST*. 2008. p. 60.

[8] Dahmann J and Morse K. High level architecture for simulation: an update. In *Distributed Interactive Simulation and Real-Time Applications, 1998. Proceedings. 2nd International Workshop on*. pp. 32–40.

[9] Schütte S. *Simulation model composition for the large-scale analysis of smart grid control mechanisms*. PhD Thesis, BIS der Universität Oldenburg, 2013.

[10] Vangheluwe H, De Lara J and Mosterman PJ. An introduction to multi-paradigm modelling and simulation. In *Proc. AIS2002*. 2002. pp. 9–20.

Table 3: Parameters used in the smart heating co-simulation use case.

Models	Parameters Descriptions	Values
thermic building	temperature setpoints of the heaters, T_{wanted}	293.15 K
	tolerance of the heaters, $bandwidth$	5 K
	electrical resistance of the heaters, R	2 Ω
	power supply voltage, U	230 V
	thermal capacities of rooms 1 to 10	112.5 kJ/K
	thermal capacities of rooms 11	600 kJ/K
	thermal conductances of the outside walls of rooms 1 and 10	2 J/K
	thermal conductances of the outside walls of rooms 2 to 9	1.25 J/K
	thermal conductance of the outside wall of room 11	7.5 J/K
	thermal conductances of the inside walls between rooms 1 to 10	3.75 J/K
	thermal conductances of the inside walls between rooms 11 and room 1 to 10	2.25 J/K
	rooms initial temperature (NB: identical for all the rooms)	293.15 K
	temperature evolution sampling period	60 s
outside temperature evolution	amplitude	5K
	offset	278.15 K
	period	1 day
	phase	$-\pi/2$
controller	consumption peaks occurrence threshold, Pow_{max}	735 W
	minimum temperature threshold, $Temp_{min}$	288.15 K
	evaluation points period (i.e. model time step)	60 s
	initial evaluation point time (i.e. evaluation points offset)	30 s

[11] Cellier FE. Combined continuous/discrete system simulation languages–usefulness, experiences and future development. *Methodology in systems modelling and simulation* 1979; : 201–220.

[12] Lara J and Vangheluwe H. AToM3: A tool for multi-formalism and meta-modelling. In Kutsche RD and Weber H (eds.) *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, volume 2306. Springer Berlin Heidelberg, 2002. pp. 174–188.

[13] Praehofer H. System theoretic formalisms for combined discrete-continuous system simulation. *International Journal of General System* 1991; 19(3): 226–240.

[14] Barros FJ. Dynamic structure multiparadigm modeling and simulation. *ACM Trans Model Comput Simul* 2003; 13(3).

[15] Esquembre F and Christian W. Ordinary differential equations. In Fishwick PA (ed.) *Handbook of dynamic system modeling*. CRC Press, 2007.

[16] Mosterman P. Hybrid dynamic systems: Modeling and execution. In Fishwick PA (ed.) *Handbook of dynamic system modeling*, chapter 15. CRC Press, 2007. pp. 1–26.

[17] Argent RM. An overview of model integration for environmental applications-components, frameworks and semantics. *Environmental Modelling and Software* 2004; .

[18] Zeigler B, Praehofer H and Kim T. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

[19] Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Proc. of CACSD '00*.

[20] Barros FJ and Zeigler BP. Model interoperability in the discrete event paradigm: Representation of continuous models. In *Modeling and Simulation: Theory and Practice*. Springer US, 2003. pp. 103–126.

[21] Quesnel G, Duboz R, Versmisse D et al. DEVS coupling of spatial and ordinary differential equations: VLE framework. In *Proc. OICMS '05*. 2005.

[22] Zeigler BP. Embedding DEV&DESS in DEVS. In *Proc. DEVS Integrative M&S Symp*, volume 7. 2006.

[23] Cellier FE, Kofman E, Migoni G et al. Quantized state system simulation. *Proc GCMS'08, Grand Challenges in Modeling and Simulation* 2008; : 504–510.

[24] Bergero F, Fernandez J, Kofman E et al. Time discretization versus state quantization in the simulation of a one-dimensional advection-diffusion-reaction equation. *Simulation* 2016; 92(1): 47–61.

[25] Kofman E. A second-order approximation for devs simulation of continuous systems. *Simulation* 2002; 78(2): 76–89.

[26] Kofman E. Discrete event simulation of hybrid systems. *SIAM Journal on Scientific Computing* 2004; 25(5).

[27] Kim YJ and Kim TG. A heterogeneous simulation framework based on the DEVS BUS and the high level architecture. In *Proc. of WSC '98*, volume 1. 1998.

[28] Mittal S, Ruth M, Pratt A et al. A system-of-systems approach for integrated energy systems modeling and simulation. In *Proc. of SummerSim' 15*. SCS/ACM, 2015. pp. 1–10.

[29] Blochwitz T, Otter M, Åkesson J et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proc. 9th International Modelica Conference*. 2012. pp. 173–184.

[30] Fritzson P and Engelson V. Modelica—a unified object-oriented language for system modeling and simulation. In *European Conference on Object-Oriented Programming*. 1998. pp. 67–90.

[31] MODELISAR Consortium and Modelica Association. Functional mock-up interface for model exchange and co-simulation – version 2.0, july 25, 2014. retrieved from https://www.fmi-standard.org.

[32] Broman D, Brooks C, Greenberg L et al. Determinate composition of FMUs for co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*. EMSOFT '13, Piscataway, NJ, USA: IEEE Press, 2013.

[33] Cremona F, Lohstroh M, Tipakis S et al. FIDE – an FMI integrated development environment. In ACM (ed.) *SAC'16*. 2016.

[34] Galtier V, Vialle S, Dad C et al. FMI-based distributed multi-simulation with DACCOSIM. In *Proc. of TMS/DEVS 15*. SCS, 2015. pp. 39–46.

[35] Barros FJ. A modular representation of asynchronous, geometric solvers. In *Proceedings of the Symposium on Theory of Modeling & Simulation*. TMS-DEVS '16, San Diego, CA, USA: Society for Computer Simulation International. ISBN 978-1-5108-2321-1, pp. 27:1–27:8. URL http://dl.acm.org/citation.cfm?id=2975389.2975416.

[36] Camus B, Bourjot C and Chevrier V. Combining DEVS with multi-agent concepts to design and simulate multi-models of complex systems (WIP). In *Proc. of TMS/DEVS 15*. SCS, 2015.

[37] Vaubourg J, Chevrier V, Ciarletta L et al. Co-simulation of IP network models in the cyber-physical systems context, using a DEVS-based platform. In SCS/ACM (ed.) *Communications and Networking Simulation Symposium (CNS'16)*. 2016.

[38] Quesnel G, Duboz R and Ramat É. The virtual laboratory environment – an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory* 2009; 17(4): 641 – 653. DOI: http://dx.doi.org/10.1016/j.simpat.2008.11.003. URL http://www.sciencedirect.com/science/article/pii/S1569190X08002165.

[39] Siebert J, Ciarletta L and Chevrier V. Agents and artefacts for multiple models co-evolution: building complex system simulation as a set of interacting models. In *Proc. of AAMAS '10*. AAMAS/ACM, 2010.

[40] Bonneaud S. *Des agents-modèles pour la modélisation et la simulation de systèmes complexes - Application à l'écosystémique des pêches*. PhD Thesis, 2008.

[41] Jennings NR. An agent-based approach for building complex software systems. *Commun ACM* 2001; 44(4): 35–41.

[42] Ricci A, Viroli M and Omicini A. Give agents their artifacts: the A&A approach for engineering working environments in MAS. In *AAMAS '07*. ACM, 2007.

[43] Camus B, Dufossé F and Orgerie AC. A stochastic approach for optimizing green energy consumption in distributed clouds. In *SMARTGREENS 2017 - Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems, Porto, Portugal, April 22-24, 2017*. pp. 47–59. DOI:10.5220/0006306500470059.

[44] Vaubourg J, Presse Y, Camus B et al. Multi-agent multi-model simulation of smart grids in the MS4SG project. In *Proc. PAAMS 15*. Springer, 2015. pp. 240–251.

[45] Chandy KM and Misra J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans Software Engineering* 1979; .

[46] Bryant RE. Simulation on a distributed system. In *Proc. of the 16th Design Automation Conf.* 1979.

[47] Fujimoto RM. Parallel simulation: parallel and distributed simulation systems. In *Proceedings of*

the 33nd conference on Winter simulation. WSC '01, IEEE Computer Society, 2001.

[48] Kofman E and Junco S. Quantized-state systems: a devs approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International* 2001; 18(3): 123–132.

[49] Tavella JP, Caujolle M, Tan C et al. Toward an Hybrid Co-simulation with the FMI-CS Standard, 2016. Research Report.

[50] Hernández-Cabrera JJ, Évora Gómez J and Cortès-Montenegro J. JavaFMI. SIANI. University of Las Palmas, Spain.

[51] Camus B, Galtier V, Caujolle M et al. Hybrid Co-simulation of FMUs using DEV&DESS in MECSYCO. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*. 2016.

[52] Moler C. Are we there yet? *Zero crossing and event handling for differential equations, Matlab News & Notes* 1997; .

[53] Petty MD and Weisel EW. A composability lexicon. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop*. 2003. pp. 181–187.

[54] Bergero F, Floros X, Fernandez J et al. Simulating modelica models with a stand-alone quantized state systems solver. In *Proc. 9th International MODELICA Conference*. 076, 2012. pp. 237–246.

[55] Floros X, Bergero F, Ceriani N et al. Simulation of smart-grid models using quantization-based integration methods. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. 096, Linköping University Electronic Press, pp. 787–797.

[56] Kofman E. A third order discrete event method for continuous system simulation. *Latin American applied research* 2006; 36(2): 101–108.

[57] Migoni G and Kofman E. Linearly implicit discrete event methods for stiff ode's. SciELO Argentina, 2009. pp. 245–254.

[58] Gilpin L, Ciarletta L, Presse Y et al. Co-simulation Solution using AA4MM-FMI applied to Smart Space Heating Models. In *7th International ICST Conference on Simulation Tools and Techniques*. Lisbon, Portugal, 2014. pp. 153–159.

Appendices

A Thermic System Model Details

In order to describe our thermic system, we need to build models for rooms to get the temperature, for walls to get the heat flow between two rooms (or between a room and the outside temperature) and for electric heaters to get the instantaneous power consumed to heat.

We use the standard library of Modelica to build our models. The thermal part of the building is built using the `Modelica.Thermal.HeatTransfer` library and the electric heater model is built with the `Modelica.Electrical.Analog` library.

Rooms are modeled as heat capacitors. Each room is seen as a volume of air with a temperature. The different influences (from the walls and from its heater) are modeled as heat flow exchanges. The behavior of the model of a room i is characterized by the equation:

$$C_i * \frac{dT_i}{dt} = Q_{in_i} + Q_{heater_i}$$

Where:

- C_i is the constant thermal capacity of the room in J/K.
- T_i is the temperature of the room in K.
- Q_{in_i} is the sum of the heat flows received from the walls connected to the room.
- Q_{heater_i} is the heat flow received from the electric heater. We consider here that it is equal to the instantaneous power consumption of the room in W –i.e. $R_i Pow = Q_{heater_i}$.

The heat flows are computed in the following way. The model of the wall determines the heat flows between the two air volumes k and l it is connected with. Note that in our case, an air volume can be a room or the outside environment. The heat flows depends on the temperatures of k and l as well as on the thermal conductance of the wall. This is represented by the following equations:

$$Q_{kl} = G_i * (T_k - T_l)$$
$$Q_{lk} = -Q_{kl}$$

Where:

- G_{kl} is the constant thermal conductance of the wall in J/K.
- Q_{kl} (resp. Q_{lk}) is the heat flow from the volume k (resp. l) to the volume l (resp. k) in J.

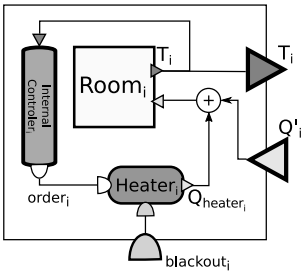


Figure 16: Heated room model "R_i"

Q_{heater_i} is determined by the behavior of the electric heater which is modeled as a basic electrical circuit with a constant voltage, an electrical resistance and a switch. This is represented by the following equation:

$$\begin{aligned} &\textbf{if } order_i \textbf{ and not } blackout_i \\ &\quad \textbf{then } Q_{heater_i} = \frac{U^2}{R} \\ &\quad \textbf{else } Q_{heater_i} = 0 \end{aligned}$$

Where:

- U is the constant voltage in V.
- R in Ω is the constant electrical resistance of each heater in the building.
- $order_i$ is a boolean representing the command of the internal controller of the heater. When it is equals to true, the heater is on.

$order_i$ is set to true when the temperature inside the room is below a minimal value, and to false when this temperature reaches a maximal value. This behavior corresponds to the conditional statement:

$$\begin{aligned} &\textbf{when } T_i \leq T_{wanted} - \frac{bandwidth}{2} \\ &\quad \textbf{then } order_i = \textbf{true} \\ &\textbf{else when } T_i \geq T_{wanted} + \frac{bandwidth}{2} \\ &\quad \textbf{then } order_i = \textbf{false} \end{aligned}$$

Where:

- T_{wanted} is the desired temperature in every room of the building.
- $bandwidth_i$ is the temperature tolerance of every heater in the building.

Each discrete port $R_i Temp$ samples the continuous temperature evolution of the room i according to the following Modelica code:

$$\begin{aligned} &\textbf{when } sample(0, period) \textbf{ then} \\ &\quad R_i\,Temp \;= T_i \\ &\textbf{end when}; \end{aligned}$$

Where $period$ is a constant interval of time in s. The Modelica function $sample(0, period)$ is used to update $R_i\,Temp$ each $period$ of time in order to represent the discrete signal regularly sent by the thermometers to the controller.

The model of a room with its heater and controller can be described in bloc diagram by Figure 16. Using this model, the whole building can be described by the bloc diagram of Figure 17. According to OpenModelica, this model is composed of 1622 equations including 11 differential equations.

B Controller Model Details

The controller maintains a set of variables $Temp_i$ and Pow_i for saving respectively the last temperature and the last instantaneous power consumption values received from the sensors of each room i of the building. Basing on these variables, the controller regularly evaluates at a given frequency if some heaters need to be disabled or enabled. If so, it sends the corresponding orders to the heaters.

The policy used to determine these orders at each evaluation point is the following:

1. The controller checks for each room i if $Temp_i \leq Temp_{min}$. If so, the controller immediately enables the corresponding heaters.
2. In order to check if some heaters have to be shut down, the controller computes the building total instantaneous power consumption Pow_{tot} according to the following equation:

$$Pow_{tot} = \left(\sum_{i=1}^{11} Pow_i\right) + \frac{U^2}{R} * N_{on}$$

With:

- N_{on} the number of heaters that have just been enabled by the controller in step 1.
- U the constant voltage of the heaters in V.
- R the constant electrical resistance of the heaters in Ω.

If $Pow_{tot} \geq Pow_{max}$, then the controller computes the number $N_{off} \in \mathbb{N}$ of heaters which have to be shut down in order to lower Pow_{tot} below Pow_{max}. The controller disables then the heaters of the N_{off} rooms having the highest temperatures. N_{off} is computed according to the following equation:

$$N_{off} = \mathsf{int}\left(\frac{Pow_{tot} - Pow_{max}}{U^2/R}\right) + 1$$

With $int : \mathbb{R} \to \mathbb{N}$ the integer typecasting function which truncates a decimal number to zero digits.

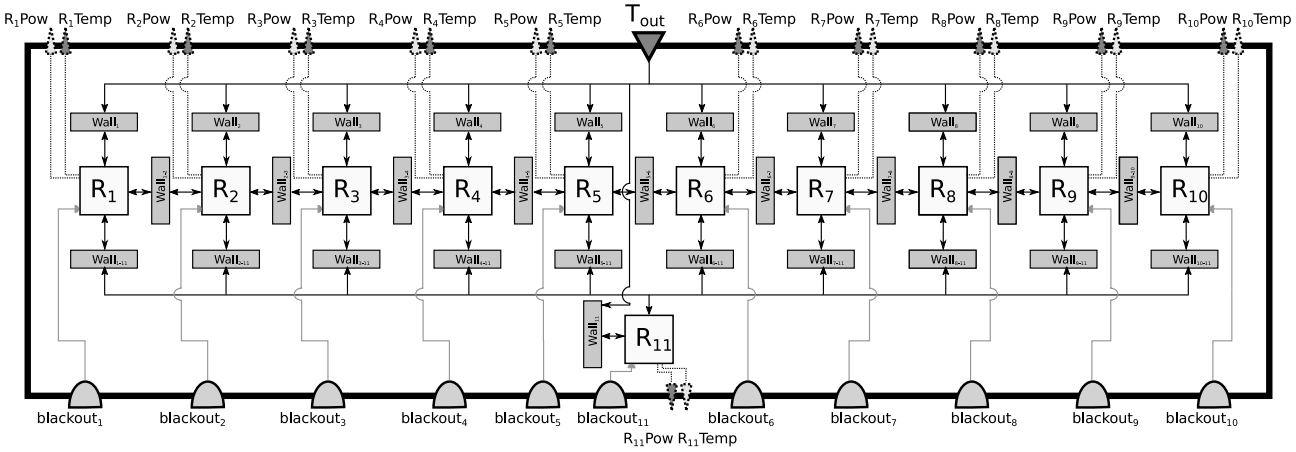


Figure 17: Building model

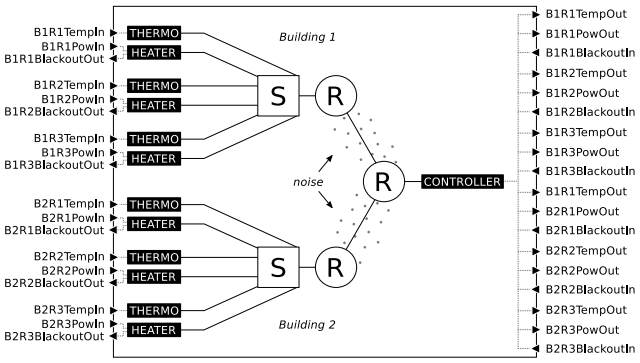


Figure 18: IP network topology, with three rooms a building, and with DEVS ports on the sides.

C Network Model Details

The IP network topology is shown on Figure 18 (with only three rooms a building instead of eleven). We consider that there is one switch (S) a building, connecting all heaters and thermometers in a same local area network. Then, each building is connected to the Internet with its own router (R). The Internet is just modeled with one big central router, and the controller is itself connected to it. Network devices are connected to external models through input and output ports (marked on the sides of the figure), for receiving and transmitting data. In this case, external models correspond to the application layer of the devices.

Heaters and thermometers can exchange measures and commands with the controller over the fake Internet, thanks to TCP or UDP connections, depending on the choice of the experimenter. Choosing TCP (reliable protocol) or UDP (unreliable protocol) is important due to the error model installed on the links between the building routers and the Internet, used for modeling some noise on the network. Experimenters can configure this error model, choosing a bit error rate (e.g. one incorrect bit for every thousand bits sent). The network model is build using the standard NS-3 component library.